Facultad de Informática,
Universidad Nacional de La Plata
Tesina de grado para obtener el grado de *Licenciado en Informática*

Meteoroid: Un MVC real para la Web

Lautaro Fernández 6394/2
Santiago Robles 6651/1

Director: Dr. Gustavo Rossi
Co-Director: Dra. Silvia Gordillo

Índice general

Índice de figuras

Agradecimientos

En primer lugar queremos agradecer a nuestras familias, en especial a nuestros viejos y hermanos quienes nos dieron su apoyo incondicional en todo momento.

A Gustavo y a Silvia por haber aceptado ser nuestros directores del presente trabajo. A los integrantes del CAG, en particular a Andrés, por sus sugerencias del presente trabajo y por ser nuestro mentor. A nuestros amigos de la facu, Pedro, Pantera, María Marta y Oscar que siempre estuvieron a lo largo del camino. A nuestros compañeros de trabajo, Nacu, Toto, Chimango, Juli, entre otros.

Ahora cada uno redactará los agradecimientos más personales, respecto a amigos y familia.

Santiago. Por sobre todas las cosas, quiero agradecerle a mis viejos por hacer siempre lo imposible para darnos todo, tanto a mí como a mis hermanos. En segundo lugar, a mis hermanos por bancarme durante tantos años. Gracias a Gabi, Raúl, Andre, Agus y sobre todo a Nacho por estar siempre. A Lau, por acompañarme durante casi toda la carrera y bancarse mi humor "particular". Por último, a mi abuela, a quien extraño un montón y que desearía que esté con nosotros en un momento tan importante de mi vida.

Lautaro. Viejos, muchísimas gracias por las incontables veces que me levantaron y me dieron el apoyo más incondicional. A mis hermanos, Lean, Loy y Malí por su constante ayuda en todas las cosas que hago. A Bel, que es mi compañera desde hace 10 años, a Santi por enseñarme a tomar mate y ser mi eterno compañero de estudios. A Leo y Germán, amigos de toda la vida.

Capítulo 1

Introducción

El desarrollo Web ha evolucionado de simple páginas estáticas a aplicaciones Web realmente complejas, alguna de ellas muy parecidas a aplicaciones de escritorio. En la mayoría de las aplicaciones los navegadores actúan como clientes livianos (o vistas) de un modelo que se encuentra en el servidor. A pesar de la evolución tecnológica de la Web, todavía no existe un mecanismo estandarizado para enviar datos o eventos desde el servidor al cliente sin un requerimiento explícito, forzando entonces a los navegadores a pedir constantemente información desde el servidor para sus actualizaciones.

Para resolver este problema un conjunto de técnicas aparecieron, las cuales permiten enviar información desde el servidor a los clientes sin un requerimiento explícito por parte del navegador. En este capítulo comentaremos las motivaciones para desarrollar **Meteoroid**, un framework con capacidades Comet [1, página 7], el cual provee entre otras cosas una forma de hacer aplicaciones Seaside [2] "vivas" con diferentes niveles de abstracción. También se verán los objetivos de **Meteoroid** y cómo se desglosará la tesis a lo largo de los capítulos.

1.1. Motivación

El comportamiento básico en el protocolo HTTP, según la RFC 2616 [3] para la comunicación entre el cliente y el servidor, es la de realizar requerimientos desde el cliente, procesarlos en el servidor y devolver la respuesta. Es decir, el servidor solamente responde a pedidos de los clientes, sin poder notificar por su cuenta algún cambio que suceda luego de terminar dicha comunicación. Esto funciona bien para aplicaciones Web en la que la latencia entre el cambio (servidor) y la notificación (cliente) no sea importante, pero en aplicaciones en donde la latencia sí importa, es necesario otro enfoque.

Esto fue de particular importancia para nosotros a inicios del año 2009, cuando

en nuestro trabajo fue necesario desarrollar aplicaciones dependientes del contexto para dispositivos móviles (PDA). El objetivo era dar un soporte gráfico para generar un modelo donde fuera posible crear aplicaciones móviles que funcionaran como servicios, con capacidades dinámicas para agregarlas y removerlas. Cada una de estas aplicaciones debía tener que conectarse a diversos tipos de recursos de cada PDA, con lo cual una aplicación Web [4] estándar quedó descartada, ya que debido al sandbox natural de un navegador Web es imposible acceder a datos locales de la misma.

La segunda aproximación fue la de generar aplicaciones de escritorio minimales para las PDAs, pero tampoco fue viable. A pesar del auge popular y a las excelentes características de hardware que los dispositivos móviles poseen hoy en día, la gran mayoría de lenguajes de programación no están optimizados a nivel gráfico, y mucho menos si son lenguajes orientado a objetos. Dejándonos entonces dos alternativas: generar vistas muy pequeñas (o pobres) o implementar el software en un lenguaje que corra nativo en el móvil (como es el caso del lenguaje C). Para nosotros esto fue un problema, porque el grupo de trabajo utiliza Smalltalk VisualWorks [5] para todas las aplicaciones y herramientas, y migrar todos los trabajos hubiera sido desgastante y posiblemente inútil/imposible en muchos casos.

La tercer alternativa surgió de pensar en cómo romper con el protocolo HTTP, y cómo aprovechar las diversas características que poseen todas las PDAs. La alternativa sería utilizar Smalltalk para el desarrollo del backend de las aplicaciones, pero no usar su soporte gráfico. En su lugar se puede aprovechar el motor gráfico que poseen los navegadores Web, embebiendo un servidor Web y desarrollar las interfaces como sitios Web. Pero si recordamos el problema del primer punto, es imposible acceder a los recursos locales de la PDA desde de las aplicaciones Web debido al sandbox. Nuestra nueva meta fue entonces, buscar una alternativa a utilizar el protocolo HTTP de la forma tradicional, donde fuera posible enviar información desde el servidor Web local hacia la página Web para poder dar exactamente la misma sensación que daría utilizar una aplicación móvil normal.

Romper con la unidireccionalidad de HTTP no es algo nuevo ni único de nuestra propuesta, sino que durante mucho tiempo se ha intentado resolver esta situación de varias maneras en el universo Web. Se comenzó refrescando periódicamente la página Web, lo que provocaba un aumento en el tráfico de la red (muchas veces innecesario) y un disgusto a nivel del usuario. Luego apareció la tecnología Ajax [6], que permitió realizar llamados asincrónicos al servidor y refrescar solamente aquellos elementos que sean necesarios (no se refresca toda la página). A pesar de que esto mejora la experiencia con el usuario y disminuye un poco el tráfico de la red, debido a que solo debe traer una parte de la página, se siguen realizando requerimientos al servidor que posiblemente sean innecesarios. Por último surgió la tecnología Comet, que propone dejar una conexión abierta para poder mandar información al cliente desde el servidor; y usando Ajax para recibir desde el cliente, se obtiene "un canal" bidireccional de información.

Al tener una comunicación bidireccional se pueden reflejar los cambios que ocurren en el servidor más fielmente en el cliente, pero existe el problema que para reflejar los cambios mencionados sólo se puede usar Javascript [7] en el navegador Web. Esto puede ser muy duro si se trabaja con muchos elementos a actualizar, existiendo además el problema de tener que manipular dos lenguajes, ya que el modelo de la aplicación va a estar realizado en un lenguaje orientado a objetos pero se necesita Javascript para poder notificar los avisos del servidor en el cliente Web.

Por tales motivos, sería óptimo extender la forma que tiene un lenguaje de objetos puro como Smalltalk a un contexto Web, unificando la forma de desarrollar el modelo y sus vistas en un mismo idioma. Si esto fuera logrado, para poder desarrollar una aplicación Web no haría falta ningún tipo de conocimiento adicional como Javascript.

La motivación de este trabajo fue la de investigar diferentes alternativas para poder obtener un Model-View-Controller (MVC) [8] en la Web, llegando a generar finalmente **Meteoroid**, una herramienta que provee tres capas de aplicaciones distintas. Cada una de estas capas es el resultado de haber desarrollado diferentes formas para resolver distintos tipos de problemas:

- La primer capa se focaliza en implementar Comet de la mejor manera posible, y permitir a la aplicación Web "hablar" con el navegador Web utilizando Javascript.
- La segunda es una forma de aislar la manipulación Javascript, permitiendo dibujar HTML tan sencillo como sea posible. En esta capa introducimos el manejo de dependencias entre un modelo y su vista.
- La capa más abstracta y más genérica es la tercera. Esta provee un variado conjunto de elementos Web (específicamente elementos de marcado de HTML) [9] para que sean dibujados de forma automática, tales como: DIVs, INPUTs, SPANs, TEXTAREAs, SELECTs, etc.

Gracias a la conjunción de estras tres capas, fue posible generar elementos de casos de prueba que permitieron comparar nuestro framework con herramientas comerciales que proveen características similares.

1.2. Objetivos del trabajo

A pesar de que contextualizar al lector con respecto a la historia del protocolo HTTP no es el propósito de nuestro trabajo, es bueno asentar una base para entender el porqué de Comet y el porqué de un nuevo enfoque hacia el mismo. El término Comet existe desde el año 2006 [10], y desde ese entonces muchos frameworks han tratado de generalizarlo y estructurarlo de diferentes formas, cada uno enfocando a sus propios problemas. Nosotros encontramos que muy

pocos habían concebido un framework que fusionara diversos aspectos como: la tecnología Comet, un ambiente de desarrollo orientado a objetos, una verdadera arquitectura MVC, y una absoluta abstracción de los problemas subyacentes a los navegadores Web y sus diversos sistemas operativos. El primer objetivo de este trabajo es entonces contar la historia que llevó a HTTP a evolucionar de su primer modelo estático a un espacio absolutamente dinámico, dando especial atención a cómo es posible hacer llegar información nueva desde el servidor al cliente (Capítulos 2 y 3).

El segundo objetivo es mostrar como fue posible generar un framework capaz de embeberse de forma sencilla a Seaside, para permitir crear aplicaciones Web "vivas" en pocos pasos. Para este propósito fue necesario adentrarse en Seaside, entender el mecanismo de renderizado y las diversas aristas relacionadas con el mismo: control flow, componentes, tasks, etc., que se encuentran especificadas en el Apéndice A. Una vez comprendido el mecanismo de trabajo de Seaside, generamos un framework para que sea acoplable a las necesidades de una aplicación Web que requiriese capacidades Comet, **Meteoroid**. Este objetivo se podría subdividir en 3 dado que nuestro framework cuenta con tres niveles de abstracción, cada uno atiende a diversos problemas.

1. La capa uno se focaliza en mostrar cómo es posible unificar Comet con Seaside, qué detalles se tuvo en cuenta y lo cómodo que es acoplarlo a cualquier aplicación realizada. Básicamente se propuso 2 metas: tener Comet y no alterar la estructura fuertemente definida a la hora de hacer aplicaciones con Seaside (como es el caso de la forma de renderizar `#renderContentOn:`). Proveemos un conjunto de configuraciones básicas en caso de no usar ciertos valores predeterminados, y son todos configurables a cada aplicación realizada en Seaside.

2. La capa dos tiene como propósito aislar al desarrollador de Javascript, tanto como fuera posible. La capa uno permite manipular el cliente a través de Comet pero solo a través del DOM tree [11][12] y llamados Javascript. Esto no es bueno, no al menos cuando lo que se busca es generar aplicaciones Web donde existen una cantidad enorme de componentes que hacen a la aplicación. Tener que manipular todo un sitio a través de Javascript sería muy costoso a nivel de debugging como de desarrollo. Por tal motivo creamos esta capa, que permite vincular un modelo de dominio con una vista Web, de manera que cada vez que el modelo cambie la misma sea actualizada a través de un bloque previamente definido por el usuario. Este bloque es puro código Seaside, cero código Javascript.

3. En la capa tres buscamos una abstracción más fuerte, hasta el punto de dejar de pensar en HTML (en algunos casos) y tratar de imaginar los componentes como tales pero ahora en la Web. Utilizando las capas uno y dos, creamos un conjunto de widgets Web que se comportan como los homónimos de

escritorio. Para realizar esto nos basamos en cómo VisualWorks trabaja con sus interfaces de usuario (GUI), y acoplamos el mismo mecanismo con el nuestro, a través de `ValueModels` [13](ver Apéndice B). Gracias a esta abstracción es posible crear componentes Web que se centren en diversos aspectos de un modelo y dejar a **Meteoroid** encargarse de cómo pintarlo y cuándo pintarlo.

1.3. Estructura del trabajo

En el Capítulo 2 se darán los fundamentos necesarios para entender la evolución de Internet, como las simples páginas que mostraban información puramente estática mutaron a complejas herramientas de trabajo y como el protocolo HTTP acompañó a estas, para cerrar con el porqué fue necesario generar una tecnología como Comet.

En el Capítulo 3 se abordará de lleno con las variadas formas de actualizar a un cliente por medio de mecanismos estandarizados, viendo en cada uno de ellos las ventajas y desventajas.

En el capítulo 4 se detallará el estado del arte con diversos frameworks que abordan Comet mostrando las capacidades, requisitos, ventajas y desventajas.

En el Capítulo 5 se explicará las diversas capas de **Meteoroid**, especificando en cada una de ellas en detalle qué fue necesario para obtenerlas, también se verán ejemplos del uso de herramientas con **Meteoroid** tanto en computadoras de escritorio como en dispositivos móviles, para cerrar con una comparativa entre nuestro framework con otros dos, viendo las ventajas de nuestra aproximación frente al resto.

Por último, la tesis finaliza con el Capítulo 6, se presentarán las conclusiones generales del trabajo realizado y se enunciarán algunos temas que han quedado fuera del mismo.

Capítulo 2

Fundamentos

Desde que Internet se popularizó, muchas de las aplicaciones que tenían por dominio el contexto de escritorio se viraron al contexto Web de forma progresiva. Esto fue posible gracias a que desde sus orígenes, HTTP fue evolucionando y consolidándose en un protocolo más robusto y abarcativo. Junto con HTTP, se sumaron los navegadores Web y Javascript, los cuales mejoraron notablemente la forma de trabajar con la Web. Hoy en día es normal desarrollar aplicaciones, que antiguamente apuntaban a un dominio de escritorio, puramente Web dado que posee los grandes beneficios como son la sencillez a la hora de debuguear y hacer el deploy, ya que sólo se debe realizar en el servidor y nada en el cliente, actualizar a nuevos requisitos, etc.

Este capítulo abarcará la estructura de HTTP, versiones y mejoras que obtuvo a lo largo del tiempo, se hablará también sobre la evolución de Internet y cómo funcionan los requerimientos y respuestas desde un navegador Web hacia el servidor. Todo esto es importante para entender porqué en este momento ciertas situaciones no son fácilmente representables con el protocolo tal cual está definido.

2.1. Evolución de Internet

Internet surgió de un conjunto de necesidades presentes en los inicios de la década del 60'. El objetivo inicial era transmitir datos entre diferentes máquinas para que pudieran interactuar con la información relevante que tuvieran. En sus inicios, Internet fue utilizada con propósitos militares para la replicación de servicios en diferentes máquinas, como forma preventiva frente a ataques nucleares. Más tarde, Internet fue usado con propósitos de investigación a través de la Advanced Research Projects Agency (ARPA)[14], una rama del Departamento de Defensa de los Estados Unidos de América. ARPA proponía estimular otras redes de computadoras mediante becas y ayudas a diversas áreas de informática

de diferentes universidades, así como también a sectores privados. Todas estos estímulos dieron fruto en 1969 a lo que se conoce como ARPAnet [15], la primer red de computadoras, la cual tenía en ese momento cuatro nodos: UCLA, Stanford Research Institute's Augmentation Research Center, UC Santa Barbara y University of Utah's Computer Science Departament.

Este primer intento de comunicación, era básicamente la ejecución de código de forma remota entre computadoras, denominada conmutación de paquetes. Unos años más tarde, en 1972, el comportamiento evolucionó generando un sistema de correo electrónico, el cual fue muy útil ya que no era necesario que haya una persona detrás de cada host para poder interactuar. Este cambio introdujo un generoso aumento en el tráfico de la red, pasando a ser la actividad predominante. En el año 1974, un grupo de investigadores propusieron el protocolo Transmission Control Protocol/Internet Protocol (TCP/IP) [16], el cual sería adoptado por el Departamento de Defensa de Estados Unidos de América para la red ARPAnet dividiéndola en dos partes: una para la ARPAnet y la otra para la MILnet (Military Network), teniendo así la primera para motivos académicos mientras que la segunda era utilizada para telecomunicaciones de tipo militar.

Internet fue evolucionando, de un mero propósito de comunicar un puñado de hosts de diferentes universidades a un gran conjunto de personas que publicaban su información en los sitios Web, y aunque estos contenidos eran información estática servían bien a su fin: propagar información de forma simple. Y fue así como Internet siguió creciendo, adoptando cada vez nuevos elementos en juego que se generaron a partir de Internet misma. El clásico ejemplo de este enfoque es el de una página que muestra información relacionada a una empresa, como puede ser la dirección, número de teléfono, horarios de atención, etc., la cual se mantiene durante muchísimo tiempo.

El problema es que los contenidos estáticos no alcanzaban para poder modelar un sitio donde la información no es invariable en el tiempo, sino que las modificaciones pueden producirse de a ráfagas en un corto período de tiempo y hasta muchas a la vez. Este tipo de sitios Web dejaron de ser un simple archivo del lado del servidor que es entregado a un cliente bajo un requerimiento explícito, para ser un archivo generado de forma dinámica con contenido actual en base a diferentes elementos (información nueva, contexto que cambió, etc.). Un ejemplo de estos, puede ser un blog donde uno o más autores publican noticias de forma cotidiana, y los lectores comentan sobre las mismas.

Gracias a estas páginas, se tuvo el control que se necesitaba para la gran mayoría de sistemas Web que existen, aunque hay ciertos elementos que este enfoque no pudo abarcar. Primero fueron páginas estáticas, luego fueron sitios dinámicos, el punto que le siguió a esto fue pensar en un aplicación Web. Cuando hablamos de aplicaciones Web debemos pensar en herramientas para usuarios finales que antes eran estrictamente implementadas para usar en escritorio, cuando ahora pasaron a ser Web. Esto trae un conjunto enorme de ventajas como son: portabilidad, actualizaciones, clientes livianos, simplicidad a la hora de debuggear y

hacer los deployments. En la próxima Sección, nos enfocaremos con mayor detalle en contar la historia de como los sitios Web se fueron convirtiendo, según el caso, en aplicaciones Web.

2.2. De Sitios a Aplicaciones Web

Previamente se comentó sobre cómo evolucionó Internet y cuáles fueron sus propósitos originales. Al principio el diseño de la Web en un contexto estático era suficiente para los objetivos de ese entonces, aunque luego se observó que este modelo tenía ciertas limitaciones que restringían generar nuevos contenidos, como son los sitios dinámicos. Esta sección se enfocará en describir el traspaso entre los sitios Web a las aplicaciones Web.

Los primeros sitios Web tenían como objetivo mostrar información relacionada a un tema específico, y hasta se los podría comparar con una tarjeta de presentación, pero en un dominio digital. Si pensamos en los primeros sitios Web (ver Figura 2.1, accedidos vía *Internet Archive Wayback Machine* [1]), notaremos que la manipulación de estilos estaba acotada a usar simplemente los tags de HTML, dado que hasta el año 1996 no existía una forma estandarizada de formatear los elementos de un sitio. Lo que era claro es que los tags HTML no fueron pensados para estilizar un sitio Web, sino para poder darle una estructura jerárquica y de importancia. Si pensamos en los tags headings (<H1>, <H2>, .., <H6>) [17] la idea era que un header de nivel 1 (<H1>) es más importante que uno de nivel 2 y que esta diferencia era implementada netamente en cada navegador Web de la mejor forma que le parezca, para que el usuario pudiera percibir que la información dentro del <H1>es de mayor relevancia que la del <H2>.

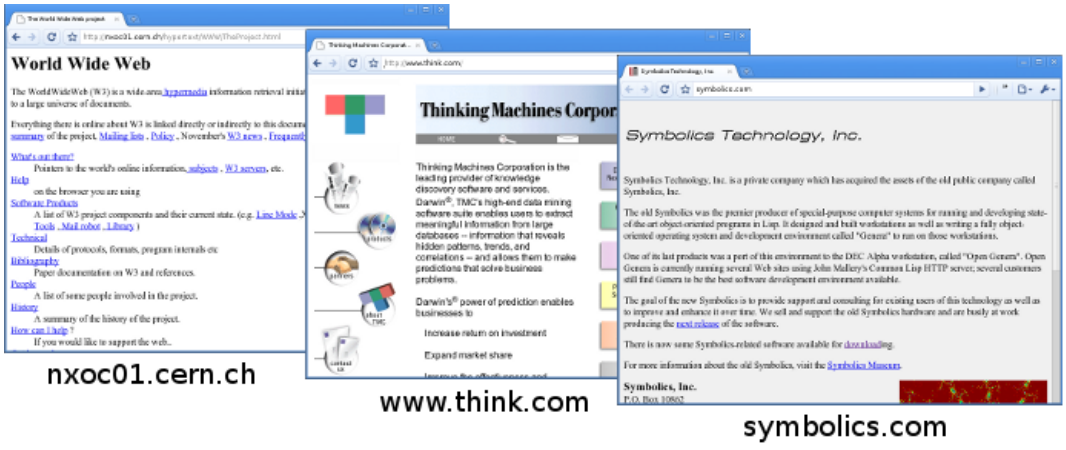

Figura 2.1: Capturas de pantalla de los primeros sitios Webs

[1] Portal dedicado a preservar históricos de los diversos sitios Web, más información en http://www.archive.org/

Esto fue un problema, hasta tal punto que se comenzaron a generar tensiones entre los desarrolladores de HTML con los desarrolladores de sitios Web, los cuales se encontraban en situaciones complicadas por no poder proveer un software a medida a sus usuarios, donde ni si quiera podían maquetar la forma de mostrar la información. En una lista de correo se puede apreciar algo de ese descontento, enviado en 1994 por Marc Andreessen el cuál decía

> In fact, it has been a constant source of delight for me over the past year to get to continually tell hordes (literally) of people who want to – strap yourselves in, here it comes – control what their documents look like in ways that would be trivial in TeX, Microsoft Word, and every other common text processing environment: "Sorry, you're screwed."[2]

manifestaba algo que era considerablemente necesario, una forma de poder definir cómo debían mostrarse los elementos ya que HTML es en sí mismo es una herramienta para definir estructuras y no estilos.

Las hojas de estilo (Cascading Style Sheets, CSS) aparecieron para solucionar este inconveniente, donde se tenía como objetivo poder cambiar tipografías y tamaños de letra, itálicas, negritas, alineamientos de texto e imágenes,colores de fondo, identificación única a elementos, y asociación por grupos, etc., fueron unas de las funcionalidades provistas por la primer definición de CSS [18], en el 1996. Dos años más tarde salió una segunda versión del mismo, CSS2 [19] la cuál define posiciones absolutas, relativas, posicionamiento arbitrarios de elementos, z-index, el concepto de tipos multimedia, etc. Esta versión CSS es la que se está usando actualmente, aunque en la revisión número 1 (CSS 2.1, generada en el año 2009) se han eliminado características de CSS1 que eran deprecated u obsoletas. Se espera que en un tiempo próximo, se libere la versión CSS3 de la cuál hay sólo un draft en este momento. Esta versión promete mayor manipulación de estilos en lo que respecta bordes, enmarcados, resizing, modularización, inclusión de fuentes no nativas, etc., aunque al ser un draft puede ocurrir que surjan nuevas o que se eliminen otras.

Junto con la evolución de las hojas de estilo, también se comenzó a abrazar la potencialidad de Javascript en el cliente. Javascript es un lenguaje interpretado (no necesita compilarse), usualmente denominado como lenguaje de scripting, estandarizado por la European Computer Manufacturer's Association (ECMA) [20] bajo ECMA-262 en el año 1997, para luego ser adoptado por la W3C [21]. La W3C estandarizó el uso de Javascript entre todos los navegadores Web en la entrada de la especificación DOM, con lo cual es posible pensar que todos los navegadores tendrán un motor de Javascript para poder ejecutar código del lado del cliente. Javascript permite manipular los elementos de la página en el

[2]De hecho, fue muy entretenido para mi tener que decirles permanentemente a una multitud (literalmente) de personas que querían – prepárense, ahí viene – manipular cómo sus documentos deberían verse tal como sucede de forma trivial en TeX, Microsoft Word, y cualquier otro ambiente de procesamiento de texto: "Lo lamento, estás fregado"

cliente, la posibilidad de generar controles para los diferentes elementos (para actuar sobre eventos de teclado, mouse, etc.), la creación y manipulación de Ajax [6], etc. Javascript acerca el comportamiento de una página a la de una aplicación de escritorio, con el uso del mismo se puede generar una página que se "sienta" como si fuese de escritorio.

Ahora entonces, es posible hablar de sitios Web estáticos, pero con una visualización personalizada que es exactamente igual en todos los navegadores Web, gracias al uso de los tags HTML más la estilización del CSS y las funcionalidades provistas por Javascript para el manejo de los controles. El problema es que los sitios Web no son puramente estáticos, ya que a pesar de que una gran parte de los sitios es siempre la misma, otra puede mutar en el tiempo. Si pensamos en una página de una empresa que provee piezas mecánicas, sería esperable que la dirección, teléfono y datos similares nunca cambien, pero el stock de productos, el staff de gente que maneja el local sí. Hoy en día suena ilógico realizar un sitio estrictamente estático, donde para actualizar cambios haya que editar el recurso HTML dentro del servidor sin ningún tipo de herramienta, y aunque esto pueda ser muy trivial para un informático no lo es para un usuario normal. En este contexto de dinamismo debemos pensar en un contenedor de información que pueda cambiar a lo largo del tiempo. Aquí debemos hablar de la evolución de las herramientas para gestionar datos como lo son las bases de datos. Estas herramientas permiten manipular información a través de un lenguajes generales como son SQL (Structured Query Language) [22], para generar nueva información, editar información existente o también eliminarla. La información es almacenada en formatos de acceso rápido, distribuida en tablas donde se tiene por objetivo agrupar información relacionada y de fácil acceso. Gracias a las bases de datos, es posible abstraerse de cómo deben guardarse los datos así como también despreocuparse de que los datos sean consistentes (por ejemplo, ante un apagón eléctrico). Cualquier sistema hoy en día que cuente con información dinámica suele usar un gestor de base de datos para persistir la información, para luego a través de herramientas como SQL mostrarlas al cliente en un formato legible.

Gracias a este conjunto de mejoras en diferentes áreas se creó el escenario para que los sitios Web sean más que un muestrario unidireccional del servidor al cliente, y pasen a tomar el lugar predominante que tienen hoy en día. Ahora es posible realizar una aplicación Web en vez de un sitio Web, cuyas diferencias residen en que ahora los clientes tienen control de los datos que hay en el modelo residente en el servidor. Si pensamos en una aplicación como un gestor de correos Web (conocidos como webmails) el cuál pueda ser accedido por cualquier navegador Web, el usuario cuenta con el poder para poder enviar correos (altas), mover el correo a diferentes carpetas (modificación), y borrar los mismos (bajas). Ese webmail puede contar con un CSS para que la página se asemeje a lo que es una aplicación de escritorio cotidiana, y utilizar Javascript para poder manipular los elementos DOM [11][12] de la página de forma tal que los cambios entre un mail y otros sean más sutiles.

Elegir un contexto Web por sobre uno de escritorio trae muchas ventajas:

- La portabilidad del software, dado que se puede acceder desde cualquier punto ya que el modelo reside en el servidor.
- La actualización automática del sistema, recordemos que el código se encuentra en un solo lugar (no hace falta redistribuir el programa a los clientes vía un medio físico).
- No hace falta instalar software en los clientes, solo acceden a una URL y el navegador Web se encarga de mostrar la aplicación.
- Son aplicaciones cross-browser y cross-platform.
- Corregir un error de código se hace menos tedioso ya que al tener el código sólo en el servidor solo debemos modificar en un solo lugar.

Recapitulando, se podría decir que existen tres pasos: estática, dinámica orientada a mostrar información, dinámica orientada a servicios. El paso actual por el que se está transitando, dinámica orientada a servicios, está enfocado no sólo a servicios sino en como el usuario "siente" a la aplicación. El modelo fundamental sobre el cual está construido HTTP define que las comunicaciones son iniciadas por el cliente cada vez que éste requiera algún recurso del servidor. Este modelo, no contempla la situación de que los cambios ocurran en el servidor por otro punto que no sea el mismo cliente que las accede, y por lo tanto no habría forma de notificar los cambios al cliente Web.

2.3. Protocolo HTTP

El protocolo HTTP (Hypertext Transfer Protocol) fue desarrollado para poder manipular información distribuida y colaborativa en un formato de alto nivel, en particular HTTP pertenece a la capa de Aplicación de la OSI [23] la cual provee la más alta abstracción de las 7 capas. En particular, la capa de Aplicación garantiza que los protocolos que se encuentren en ella, deben garantizar una comunicación segura entre aplicaciones que corren en hosts remotos.

Este protocolo ha sido ámpliamente usado por la World Wide Web (WWW) desde los inicios de la década de los 90', utilizado en aquel entonces lo que se conoce como HTTP/0.9. Esta primera versión del protocolo fue pensada para enviar información cruda a través de Internet entre navegadores Web y servidores, y a pesar de ser funcional también era extremadamente limitada, puesto que el único comando que el cliente Web podía realizar era el "GET". La información devuelta por el servidor HTTP era un archivo de texto plano cuyo contenido era un documento HTML. La forma de trabajo que tiene este protocolo, se describe como: el navegador Web realiza una conexión al servidor pidiendo (GET) un

archivo alojado en él, el servidor busca el mencionado archivo y lo entrega como respuesta del cliente, el servidor cierra la conexión. Notar que por cada requerimiento nuevo, el cliente debe realizar una nueva conexión al servidor, esto provoca muchos problemas que serán comentados y desarrollados en Sección 2.4. Como detalle peculiar, nunca se lanzó un RFC [24] para este prototipo de HTTP, así como tampoco tuvo una versión formal, es por eso que se la conoce como "HTTP versión 0.9" o "HTTP/0.9". Lo más parecido a un RFC para esta versión se detalla en el documento *asImplemented* [25] del World Wide Web Consortium (W3C), el cual define las características básicas de la conexión, el formato por el cual los datos viajan a través del GET, etc. La funcionalidad provista por HTTP/0.9 permitió que la WWW evolucione en muchos aspectos y se convirtiera cada vez más popular, permitiendo generar nuevos tipos de contenidos.

La versión estable del protocolo, HTTP/1.0, nació para resolver el problema de la limitada capacidad de HTTP/0.9 para poder trabajar con diferentes tipos de archivos, dado que el soporte de HTTP/0.9 era únicamente para documentos de hipertexto y texto plano. La solución fue tomar prestada la idea de Multipurpose Internet Mail Extensions (MIME) [26][27], donde existe un "header" (o encabezado) y el "body" (información relevante a procesar), el cual provee la capacidad para enviar prácticamente todos los datos existentes y posibles. Esta versión estable fue publicada a mediados de la década del 90 bajo la resolución RFC 1945 [28], no sólo explicitando los tipos MIME soportados sino también dando una detallada explicación de como debe producirse la comunicación entre un servidor de HTTP y un cliente del mismo protocolo. El mecanismo por el cual inician una comunicación entre un servidor y un cliente es denominado "hand shaking", en donde ambos extremos abren una conexión TCP por la cual van a enviar y recibir datos. Una vez establecido el canal seguro de transmisión, ambas partes siguen un estricto protocolo de envíos de mensajes que aseguran la correcta transferencia de información. Este énfasis no fue en vano, ya que la versión previa de HTTP no definía de forma explícita el formato de mensajes que se debían emitir de ambos lados.

Esta resolución de HTTP/1.0 no fue suficiente para ciertas características de la Web, y rápidamente se encontraron nuevas optimizaciones que eran posibles de realizar para mejorar el protocolo. En el 1997 se lanzó la versión de HTTP/1.1 bajo la RFC 2068. Esta versión contempla ciertos problemas de rendimiento (más que nada, en velocidad) con respecto a su antecesor y las corrige, como por ejemplo la utilización de una misma conexión TCP/IP entre cliente y servidor para atender varios requerimientos, un mejor cacheo en el servidor, el uso de proxies, un mejor sistema de seguridad, etc. La última versión publicada de HTTP/1.1 fue en 1999 bajo la RFC 2616.

Independientemente de la versión de HTTP en la que estemos trabajando (1.0 o 1.1) ambas comparten ciertas características que resultan interesante destacar: ambos son protocolos sin estado (stateless), hacen uso de las conocidas "cookies" (HTTP State Management System, RFC 2109) e implementan el mismo meca-

nismo de comunicación entre el cliente y el servidor Web. Se denomina a HTTP como un protocolo stateless porque el servidor no posee información sobre los clientes, cada vez que devuelve información a un requerimiento solicitado destruye el contexto donde fue creado, liberando recursos. Al ser HTTP un protocolo que no posee estado, se requiere algún método adicional para poder agregar comportamiento a los requerimientos aislados que puede generar un cliente Web. Si un cliente no tiene forma de dar contexto a su request, el servidor no puede entregar una respuesta adecuada para ese cliente, dado que no tiene más información que el recurso que el cliente quiere acceder del servidor Web. Es por esto que se comenzó a utilizar las cookies, las cuales permiten identificar y darle un contexto a cada request del cliente Web. Con respecto a la comunicación entre cliente y servidor, el protocolo HTTP se diseñó para que cada vez que el cliente quiera algún objeto del servidor, el cliente deba solicitarlo de manera explícita. De no ser así, el servidor no tiene la capacidad de llevar información nueva al cliente. Esto, a pesar de que puede ser algo cotidiano, imposibilita ciertas formas de interactuar entre los clientes Web y el servidor.

2.4. Necesidad de un nuevo enfoque

Previamente se comentó el problema que existe en la arquitectura actual de HTTP, el cual imposibilita actualizar el cliente Web sin que éste realice un requerimiento hacia el servidor por nuevos datos. En el protocolo HTTP la comunicación entre el servidor y el cliente Web fue pensada (y sigue siendo actualmente según la RFC 2616) para que el servidor respondiera a un requerimiento sólo si antes hubo un cliente que pidiera por algún recurso. Esto implica que el servidor no tiene la capacidad para enviar nueva información hacia el cliente, si es que este último no se lo pidió de forma explícita al servidor.

En el año 1995 Netscape vislumbró este detalle y presentó un documento llamado "The Great Idea" [29] que trataba de modo teórico la discapacidad de enviar datos desde el servidor al cliente HTTP, proponiendo los mecanismos de "Client Pull" y "Server Push". El modelo de Client Pull se basa en requerimientos hechos por el navegador Web para obtener información nueva desde el servidor. Cada vez que el cliente Web quiere refrescar información de la página, debe generar un requerimiento para esa información para luego tratarla de la forma que desee. Por el otro lado, la técnica de Server Push deja una conexión abierta entre ambos extremos, habilitando al servidor para enviar nuevos datos cuando requiera sin que el navegador Web los requiera explícitamente.

A continuación se describirán ambos métodos en profundidad, explicando las técnicas que pertenecen a uno u otro método. A su vez, se tomará como ejemplo de cada técnica, el desarrollo parcial de un chat, el cual consta de una lista de usuarios, una lista de mensajes recibidos y un input por el cual se ingresará el texto que se desee enviar. El código HTML generado tendrá la siguiente forma

```
<html>
    <head>
        ...
    </head>
    <body>
        <div id="usersList">
            <em>Users:</em><br/>
            <select id="selectUsersList" size="19" name="4
                ">
                <option>User1</option>
                ...
            </select>
        </div>
        <div id="content">
            <div id="room">
                <textarea id="messages"/>
            </div>
            <div id="userAndText">
                <textarea id="message"/>
            </div>
        </div>
    </body>
</html>
```

el cual junto con un CSS que proveerá el estilo se verá de la siguiente forma

Figura 2.2: Ejemplo de chat Web

2.4.1. Client Pull

Client Pull es la forma más común de realizar una conexión entre un navegador Web y un servidor, donde cada vez que el cliente necesita algún dato del servidor, realiza una nueva conexión donde más tarde el servidor colocará los datos para él. Este mecanismo ha probado ser exitoso para una infinidad de casos, aunque como

veremos a continuación no es la mejor forma para realizar una aproximación del escritorio a la Web.

En esta sección vamos a presentar tres diferentes técnicas para actualizar información en los navegadores Web utilizando la técnica de Client Pull. La primera está basada en refrescar la página entera mientras que la segunda y tercera se basan en actualizar puntos específicos de ella. Para tener una idea generalizada de cómo es Client Pull ver la Figura 2.3.

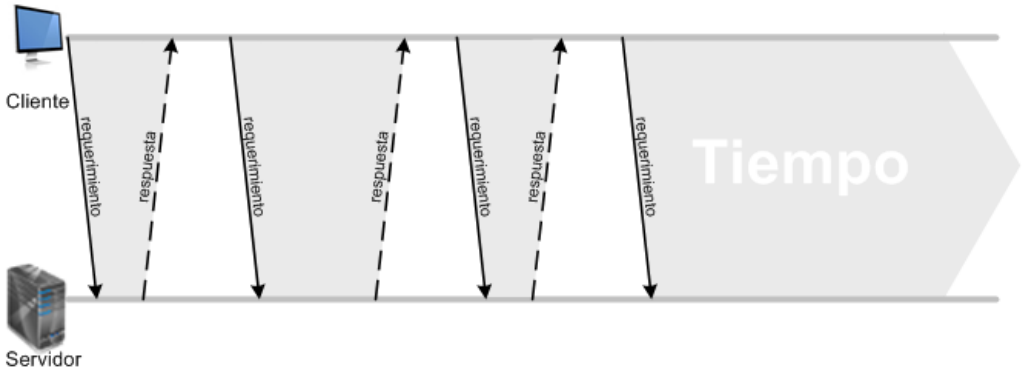

Figura 2.3: Diagrama Client Pull

Refresco Total

Esta técnica es conseguida insertando un META Tag [30] en el elemento HEAD del HTML, de esta forma se fuerza un refrescado completo de la página. Junto con el META Tag debe definirse un tiempo límite (timeout), el cual determinará el tiempo máximo que ocurre entre que se muestra la página y se produce un nuevo requerimiento desde el navegador Web para volver a mostrar la página. Si el timeout se define en un número apropiado con respecto a los cambios del modelo de negocios, la página reflejará el contenido del servidor de forma apropiada. Por ejemplo, se podría tomar el ejemplo del chat y agregando la siguiente línea en el HEAD, se actualizará cada un segundo toda la página, y el servidor la redibujará con la nueva información, ya sea, con un mensaje nuevo o algún usuario que haya ingresado o se haya ido:

```
<head>
    ...
    <meta http-equiv="refresh" content="1">
</head>
...
```

Como se puede apreciar, resulta sencillo agregar el mecanismo para actualizar toda la página, pero el servicio provisto es bastante malo, ya que debe refrescar

toda la página sin importar si hubo cambios o no, o si sólo hubo cambios en un sector. Además, el refresco de la página puede ser demasiado tarde o demasiado prematuro para los cambios que ocurren en el modelo.

Ajax y Javascript's Timers

Otra forma de implementar un mecanismo de Client Pull es combinando Ajax [6] y relojes de Javascript [31]. Ajax permite al navegador Web realizar llamadas "silenciosas" al servidor para pedir por nueva información cuando este lo requiera. Notar que estas llamadas al servidor son asincrónicas lo que habilita al navegador Web seguir ejecutando otros códigos de Javascript (los llamados Ajax son no bloqueantes). Con las funciones reloj de Javascript (setTimeout o setInterval) se puede escribir un loop infinito para que periódicamente el navegador Web realice un requerimiento al servidor por nueva información, procesarla y decidir luego si parte de la página debe modificarse para notificar los nuevos cambios. Esto último se logra gracias a la manipulación de elementos del árbol DOM del HTML. El requerimiento "silencioso" se realiza mediante un XMLHttpRequest [32], que básicamente devuelve información del servidor al script del navegador Web el cual luego la puede procesar.

Retomando el ejemplo del chat, en este caso no se necesitará de un refresco total de toda la página, si no que se pueden definir funciones Javascript que refresquen determinados fragmentos de interés. A continuación se mostrarán dos funciones, `updateMessages()` y `updateUserList()`, que actualizarán la lista de mensajes y la lista de usuarios respectivamente cada una determinada frecuencia:

```
function updateMessages(){
    // En esta funcion se realiza un llamado Ajax
       para pedir por los nuevos mensajes
}
setInterval( "updateMessages()", 1000 );
function updateUserList(){
    // En esta funcion se realiza un llamado Ajax
       para pedir por la lista de usuarios
       actualizada
}
setInterval( "updateUserList()", 1500 );
```

La función setInterval toma como parámetro una función a ejecutar y un tiempo, para que una vez finalizado esa espera ejecute la función antes mencionada. Se observa también en el ejemplo, que se puede tener diferentes funciones con distintos intervalos de actualización para refrescar determinados fragmentos de la página (los mensajes pueden actualizarse más rápido que los logueos de los usuarios y por ese motivo se refrescarán más rápidamente). Pero aún con esta

mejora, los refrescos pueden seguir siendo demasiado rápidos o demasiados lentos dependiendo de la actividad de la aplicación.

Long Polling

Esta propuesta [1, página 41] presenta una técnica la cual consiste en hacer un requerimiento al servidor, pero este último en vez de devolver una respuesta completa y cerrar el canal de transmisión, si no posee ninguna información para devolver, deja el canal abierto para futuro uso. Entonces, cuando el servidor lo crea necesario envía la nueva información al navegador Web para luego cerrar la conexión. Una vez que la información es alcanzada por el navegador Web, inmediatamente abre otra conexión al servidor y luego procesa los datos que arribaron (ver Figura 2.4). Esta reconexión del lado del navegador Web se realiza para que no haya un espacio grande entre "canales abiertos", y de tal forma el servidor puede enviar cuando quiera la información nueva al navegador Web, emulando de alguna manera lo que se denomina un canal persistente. Esta técnica es similar a Ajax y Javascript's Timers, pero se comportan distinto a nivel de requerimientos HTTP: Long Polling va a realizar un requerimiento y hasta que el servidor no devuelva una respuesta, no va a realizar otro requerimiento. Por el otro lado, Ajax y Javascript's Timers está realizando constantemente requerimientos para consultar al servidor por nuevos cambios.

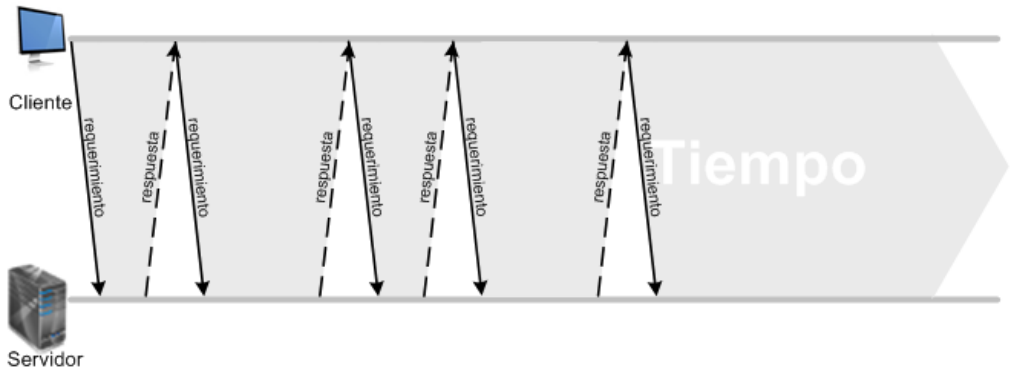

Figura 2.4: Diagrama de funcionamiento de Long Polling

La segunda y tercer forma de implementar Client Pull, tienen algunas ventajas sobre el primero, ya que ambas utilizan un requerimiento del tipo XMLHttpRequest. Esto implica que el comportamiento de la página es siempre amigable al usuario:

- Como los requerimientos XMLHttpRequest son no bloqueantes, el motor de Javascript del navegador no se queda paralizado esperando por una respuesta

- El requerimiento XMLHttpRequest puede pedir información asociada a elementos específicos, y gracias a eso la carga de tráfico se reduciría a los elementos que se usan.
- El usuario no nota el refrescado de la página (flickering, similar a un parpadeo) que se produce con la técnica de Refresco Total, ya que al manipular el árbol DOM de la página pueden editar la información que crean necesarios (en este caso no ocurre flickering)
- Reducen procesamiento, tanto en el cliente como en el servidor dado que solo se trabajan con datos específicos

Si volvemos al ejemplo del chat y centrándonos en la actualización de la lista de mensajes (la lista de usuarios es muy similar), vamos a tener una función (`updateMessages()`) que creará un requerimiento Ajax, el cual el servidor dejará abierto hasta que posea información para devolver. Una vez recibido del lado del cliente y procesado (realizando un llamado a la función `renderMessages(messages)`), éste volverá a crear un nuevo requerimiento el cual estará escuchando por nuevos cambios.

```
function renderMessages(messages) {
    //En esta funcion se actualiza la lista de
       mensajes
}
function updateMessages() {
    var httpRequest = new XMLHttpRequest();
    httpRequest.open("GET", "http://myHost/
       streaming.response");
    httpRequest.oncomplete = function(xhr){
                if (xhr.status == 200)
                   renderMessages(xhr.responseText)
                   ;
                updateMessages();
          };
    httpRequest.send();
}
```

Como se ve en el ejemplo, utilizar esta técnica no es muy complejo, pero este tipo de técnicas sobrecargan la red en momentos de frecuencias de actualizaciones muy elevadas, debido a que crearán y cerrarán demasiadas conexiones.

2.4.2. Server Push

Una aproximación completamente diferente para enviar información del servidor al navegador Web sin que este realice un requerimiento explícito es la técnica

de Server Push. Esta técnica tiene la ventaja de mandar datos sólo cuando el servidor crea necesario, evitando arribos de respuesta en el navegador Web fuera de lugar o lentos, así como también se evitan los envíos innecesarios. Sin embargo, Server Push implica dejar una conexión constantemente abierta (generalmente, un socket [33][34] abierto) por cliente, lo cual, dependiendo de las capacidades del servidor puede conducir el mismo a un mal funcionamiento. Para evitar esto último, se pueden usar diferentes opciones como clusters de computadoras [35] o balanceo de carga [36].

Esta diferencia entre Client Pull y Server Push se hace más importante cuando se desarrolla aplicaciones Web donde la relación que existe entre los cambios y el tiempo que ocurre entre ellos no pueden ser anticipados, o en aquellos casos donde grandes ráfagas de cambios son seguidas por largos tiempos de inoperabilidad. En esos casos usar Client Pull con un tiempo corto de timeout producirá una aplicación con buenos tiempos de respuesta, pero generaría mucho tráfico innecesario y procesamiento de CPU en los momentos donde no haga falta pedir por nueva información. Por el otro lado, estipular un tiempo largo entre requerimientos provocaría que se pierdan determinados cambios producidos en el servidor, específicamente en las partes donde hay ráfagas de cambios (una comparación interesante se puede observar en [37]). Ejemplos claros de este tipo de aplicaciones, donde el modelo de negocios tiene tiempos de cambios no estipulados y altamente cambiantes, son Chats, Tweets, InstantMessaging, etc, donde muchas líneas de texto pueden ser intercambiadas en segundos, cuando en otros casos pueden pasar minutos o inclusive horas hasta que un nuevo cambio ocurre en el servidor. En esta situación la técnica de Server Push entrega los cambios a tiempo, sin generar procesamiento innecesario o carga de tráfico innecesario, como se ve en la Figura 2.5.

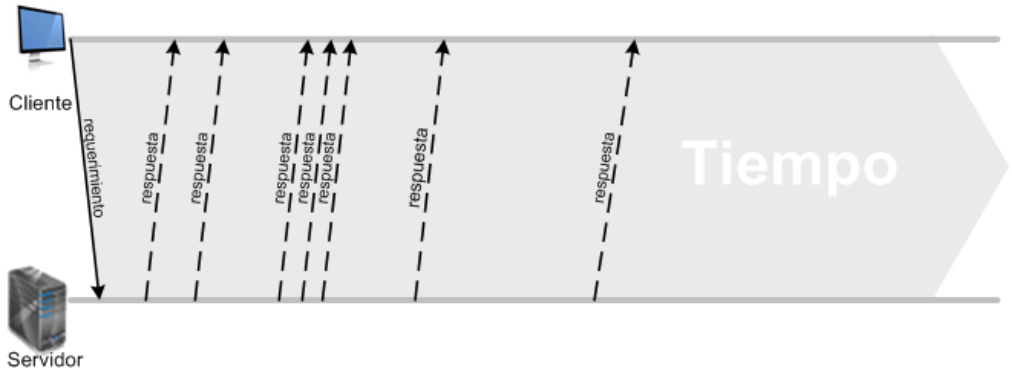

Figura 2.5: Diagrama Server Push

Combinando la técnica de Server Push (la cual permite enviar datos del servidor al navegador Web) con Ajax (que permite enviar datos del navegador Web al servidor) da origen al concepto de Comet [1, página 7]. Comet es una forma

de intercambiar datos entre servidor y el navegador Web, especialmente enfocado en cambios ocurridos por el servidor. Desafortunadamente, Comet (el cual es actualmente un conjunto de técnicas para comunicar servidores y navegadores Web) no es un estándar, y los desarrolladores tienen que lidiar con las diversas complicaciones dependiendo de los diferentes tipos de navegadores Web que existen (Mozilla Firefox, Opera, Chrome, Internet Explorer, etc.) para lograr armar una aplicación con Comet. En el estado actual existen dos tendencias principales para lograr un Server Push: usando plugins [38] en los navegadores Web o combinando HTML y Javascript.

Comet: Basado en Plugins

Las implementaciones basadas en plugins fueron las primeras en lograr con éxito una implementación de Server Push. La más conocida en esta área es utilizando los Java Applets, los cuales mantienen una conexión TCP persistente entre el servidor y el navegador Web. Otros ejemplos basados en plugins son utilizando Flash [39], Silverlight [40] y OpenLaszlo [41].

A través de la adición de un Applet a la página se podría implementar el ejemplo del chat, el cual va a crear una conexión TCP dedicada para recibir toda la información nueva que el servidor quiera enviar. Cuando llegue la información el Applet la procesará y generará un Javascript que actualice tanto la lista de usuarios como la lista de mensajes. La esquematización de la página quedará como se muestra en la siguiente imagen:

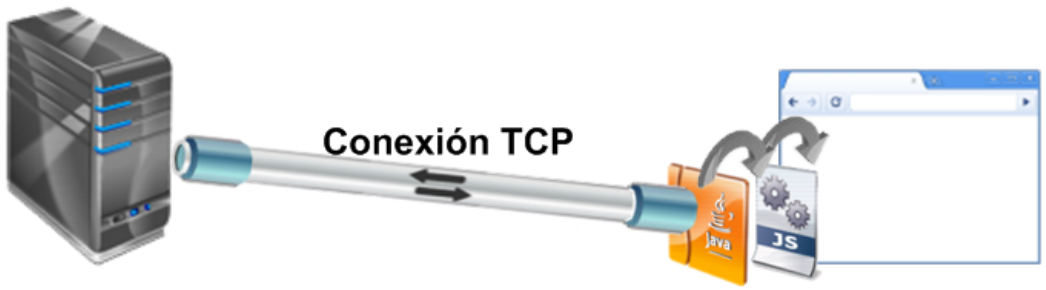

Figura 2.6: Chat usando Applet

Existen varios problemas con todas las opciones Comet basados en plugins:

- No son parte de los navegadores Web por defecto, requiriendo una instalación específica en cada cliente.
- Alguno de ellos restringen ciertos sistemas operativos (por ejemplo, Silverlight sólo funciona para Windows).
- En algunos casos, estos plugins son software propietarios, lo que limita aún más el uso para cualquier persona.

Por el otro lado, si la implementación de Comet usase solo componentes estándares (como por ejemplo Javascript), la aplicación Web funcionará sin ningún problema en los navegadores Web que son W3C-Compliant [42] (navegadores que siguen los estándares Web).

Comet: Basado en estándares Web

La alternativa para evitar los plugins en los navegadores es combinar HTML y Javascript, evitando cualquier requerimiento especial en cada navegador Web. De hecho, usando elementos del protocolo estándar, Comet también funciona en navegadores Web "más rudimentarios" (como los que se encuentran en dispositivos móviles). En esta área, encontraremos diversas técnicas para diferentes navegadores Web, que no serían capaces de implementar de no ser por los servidores con capacidades de streaming. En la Sección 3.1.1 se explicará en mayor profundidad qué es un servidor streaming, pero a grandes rasgos, un servidor con dichas capacidades tiene la posibilidad de enviar información al cliente mientras se va generando la respuesta. En un inicio estos servidores fueron útiles para poder entregar datos del estilo multimedial (grande fotos, audios, videos, etc), donde el tamaño a primera instancia no puede ser determinado.

Comet aprovecha este tipo de servidores para entregar una respuesta infinita, donde puede ir volcando la nueva información a medida que se crea y a través de la modificación de los elementos DOM de la página manipula el comportamiento y contenido de la misma de forma dinámica.

En el caso del chat, en vez de tener un objeto Applet como se comentó en la subsección anterior, tendremos un objeto Javascript que será el encargado de mantener la conexión Comet con el servidor y a su vez actualizar tanto la lista de usuarios como la lista de mensajes. Por lo tanto, la página Web quedará como se muestra en la siguiente imagen:

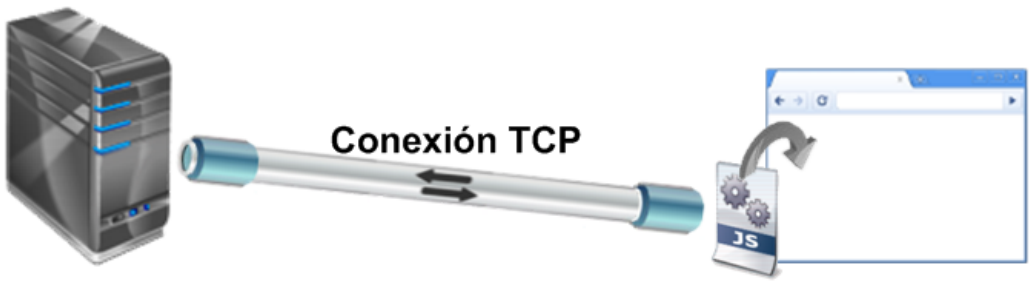

Figura 2.7: Chat usando Javascript

2.5. Conclusión

Luego de haber presentado un contexto general de cómo evolucionó el protocolo HTTP según las diferentes RFC, el desarrollo y auge de Internet, y la vasta extensibilidad del mismo, se observó que el mecanismo actual sobre el que trabaja HTTP está diseñado para no permitir manipular el navegador una vez cerrada la conexión. Es por esto que se investigó sobre diferentes alternativas al respecto que tratan de solucionar este impedimento, en particular hablamos sobre las metodologías Client Pull y Server Push, cada una con varias implementaciones, llegando en el caso de Client Pull a Long Polling y para Server Push a Comet utilizando los estándares.

En particular consideramos que Long Polling no es una buena técnica en comparación con cualquiera de Server Push, ya que Long Polling falla en determinados casos tales como:

- Por cada respuesta del servidor, se cierra la conexión actual para realizarse otra nueva de forma casi instantánea. Esto agrega más retrasos y sobrecarga a la red, y en casos de altas transferencias de datos el navegador Web se la pasaría creando conexiones.
- El servidor debería implementar una cola de respuestas para cada cliente, ya que una vez enviado un conjunto de datos, debe esperar a que el cliente realice una nueva conexión para poder volcar los datos en ella.
- Los retrasos que ocurren por cerrar y abrir conexiones para cada dato, junto con la cola de mensajes del servidor pueden provocar que los datos realmente nuevos y relevantes tarden en llegar así como también en visualizarse.
- Nosotros no lo consideramos como una alternativa real a Comet, dado que la conexión entre el cliente y el servidor no es "persistente".

También pensamos que la opción de utilizar técnicas que utilicen plugins no es buena debido a que:

- Se requiere la instalación de un plugin para el funcionamiento correcto de la aplicación.
- No siguen los estándares y por lo tanto hace que el desarrollo de este tipo de aplicaciones sea más complejo ya que debe tomarse diferentes precauciones de acuerdo al navegador que se esté utilizando.
- Atenta contra el soporte sobre la totalidad de los navegadores Web ya que los plugins suelen no estar soportados en toda la variedad de navegadores ni en todos los sistemas operativos.
- La existencia de plugins en dispositivos móviles el prácticamente nula.

Por lo tanto la opción de utilizar Comet utilizando estándares resulta ser la mejor opción basándonos en que:

- No se necesita instalar nada sobre el navegador Web para que la aplicación funcione correctamente.
- Al basarse en estándares facilita el desarrollo, ya que la mayoría de los estándares están implementados en los navegadores, incluyendo los que son utilizados en dispositivos móviles.
- No hay sobrecarga de tráfico en la red debido a que no se crean conexiones por cada respuesta, como sí sucedía con Long Polling.
- No se debe crear ninguna cola de respuestas ya que ni bien ocurre un nuevo evento al servidor es enviado por la conexión persistente.

En el próximo capítulo se abordará en mayor detalle las variadas formas de implementar Comet sin la necesidad de instalar adicionales en los navegadores Web.

Capítulo 3

Desglosando Comet

En el capítulo anterior se detallaron los motivos por los cuales se considera que Comet es la mejor solución para el problema de realizar cambios en las páginas que ya fueron entregadas al navegador Web, mientras no haya un estándar que sea implementado por todos los navegadores. Esta sección se focalizará en detallar Comet, explicando qué tipos de requerimientos son necesarios en el servidor como en el cliente, así como también sus limitaciones. Luego se detallarán las diversas técnicas que existen para poder implementarlo y para cada una de ellas se describirán sus características, explicando las ocasiones en las que conviene utilizar alguna de ellas en particular.

3.1. Requerimientos

Para que un servidor tenga capacidades Comet, debe atender al menos dos temas de importancia, los cuales son las capacidades de streaming (poder enviar datos del servidor al navegador Web de forma parcial) y la capacidad de atender múltiples conexiones (que sucedería cuando varios clientes se conectan a una aplicación al mismo tiempo). Nos enfocaremos en explicar ambos contenidos a continuación.

3.1.1. Servidor con capacidades streaming

Una posible implementación de Comet con HTML y Javascript es usar streaming [43]. Antiguamente, cuando un servidor recibía un requerimiento se creaba una respuesta y se enviaba de vuelta al navegador Web. Una vez que la respuesta llegaba de forma completa, el navegador Web se encargaba de renderizarla de la forma apropiada, algo tedioso a la hora de navegar la Web. A medida que los servidores Web evolucionaron empezaron a soportar streaming, que los habilitaba a

enviar respuestas a los navegadores Web en partes (chunks) mientras iban siendo generadas, dejando a estos últimos mostrar los datos parciales del requerimiento. En la Figura 3.1 se ve un ejemplo de como funciona.

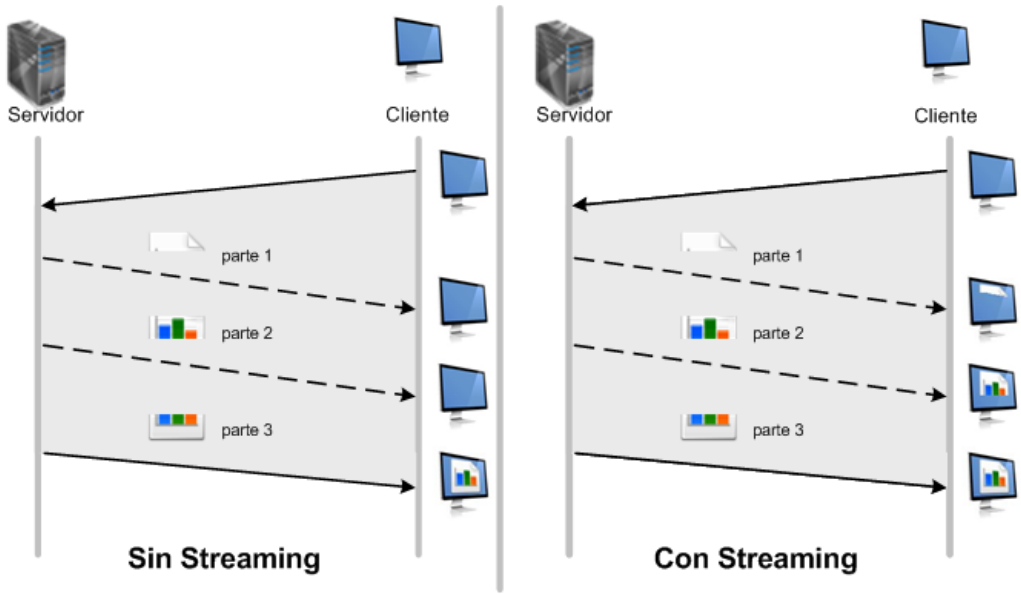

Figura 3.1: Servidor con y sin streaming

Streaming mejora notablemente la experiencia de usuario, dado que la información en la página puede aparecer sin la necesidad de que la misma se haya descargado de forma completa, dando la impresión al usuario que la página en sí es cargada más rápidamente, cuando en realidad pueden haberse bajado solo algunas estructuras del documento, algunas librerías Javascript y el CSS. La capacidad de streaming es también muy útil para devolver largas respuestas, tales como recursos multimedia (imágenes grandes, videos, audio, etc.) los cuales en otro contexto tardarían muchísimo más tiempo para poder mostrarse (no se podría ver la imagen, hasta que no se haya bajado el contenido entero), claros ejemplos hoy en día son los reproductores de música [1], de video [2], radios on-line, etc.

En el escenario de Comet, streaming se usa para implementar una comunicación bilateral entre servidor y navegador Web. El truco está mantener un canal abierto, forzando al servidor a nunca cerrar la respuesta al cliente y usar ese canal para enviar datos cuando requiera. Cuando nuevos eventos en el servidor deban ser reflejados en el navegador Web, son enviados por la respuesta abierta. El navegador Web puede entonces, ir capturando las respuestas parciales para luego con Javascript procesarlas, y actualizar los contenidos del árbol DOM si requiriese.

[1] Una página muy conocida es www.goear.com

[2] Entre los sitios de video se encuentran www.youtube.com, www.vimeo.com, etc.

Como dato particular, en el 1999 se usó una técnica de streaming diferente. Esta es muy parecida a la forma general de streaming, pero en este caso particular Ka-Ping Yee (creador de la técnica), utilizaba una imagen GIF infinita [44] para crear un chat. La idea básica era entregar una imagen GIF que nunca terminara, y que cada vez que un usuario del chat mandara un nuevo mensaje, el servidor procesara el mismo como una fila nueva del GIF, y mandarla a los navegadores Web para que posteriormente fuera dibujada en cada uno. Si se observa la Figura 3.2 se ve que en un inicio solo se encuentra logueado *lautaro*, luego ingresa *santiago* al chat y se mantiene una breve conversación entre ambos "participantes".

Figura 3.2: Imagen del chat usando un GIF "infinito"

A pesar de que esta alternativa es bastante limitada, dado que no hay manipulación DOM a través de Javascript y que los cambios ocurren en un sandbox, es una buena forma de mostrar las capacidades que ofrece streaming.

3.1.2. Múltiple conexiones

Los recursos dispuestos por una computadora son finitos, tanto sea para los clientes Web como para los servidores Web. Los problemas que podemos encontrarnos con Comet están altamente relacionados a los recursos y limitaciones implícitas del HTTP.

Si nos enfocamos en el navegador Web, podemos decir que el RFC de HTTP1.1 especifica que se debería poder realizar 2 conexiones simultáneas a un servidor como máximo. Esto último debe entenderse como que no pueden ocurrir 3 o más requerimientos HTTP de un cliente Web a un servidor bajo la misma IP/servidor de dominio, ya que en muchos casos quedaran en cola de espera. Por ejemplo, si dentro de una misma página realizamos 3 llamados Ajax por un objeto del servidor, los cuales tienen como destino la misma IP/servidor de dominio, bajo un navegador como Firefox v.2.0 solo se realizarán dos requerimientos, una vez cerrada la respuesta de uno, el tercero tomará lugar. Navegadores Web más nuevos, permiten más conexiones en pro de optimizar la carga de imágenes y otros documentos afines a un sitio web. En particualar:

- Internet Explorer 7 (o inferior), 2 conexiones

- Internet Explorer 8, 6 conexiones
- Firefox 2.0 (o inferior), 2 conexiones
- Firefox 3.x, 15 conexiones
- Google Chrome 5, 8 conexiones
- Opera, 8 conexiones

Teniendo en cuenta que para realizar una *conexión Comet* se aprovecha de un canal "infinito" de información, se suele utilizar un segundo canal para introducir los datos al navegador Web. Cada vez que un navegador Web entra a un sitio con Comet, realiza un primer requerimiento HTTP para obtener los datos del sitio, y luego un segundo para habilitar el canal de datos desde el servidor (la *conexión Comet*). En particular, siempre se suele hacer un segundo requerimiento HTTP para evitar problemas visuales en el navegador, dado que si se usara el primero como la *conexion Comet* para enviar datos, el mismo sería una respuesta infinita. Esto provocaría que el navegador mostrara la página cargando (este detalle será explicado en breve), generando una sensación de incompletitud al usuario final.

Por el lado del servidor, nos podemos enfocar en al menos dos características: alta carga en memoria y limitaciones de sockets.

Alta carga en Memoria. La memoria está estrictamente relacionada con la cantidad de elementos que se van a utilizar. Debemos tener en cuenta que por cada nueva conexión, debe haber al menos un socket que la acompañe. Dependiendo de cada sistema operativo, los sockets pueden tener tamaños distintos, variando los datos que fueran a alocar en memoria, los cuales son: información de control y estado, flags, y por su puesto los datos que van a viajar en el socket. Los sockets pueden ser TCP o UDP, siendo este último más chico en memoria, pero más escueto en capacidades de confiabilidad, verificación y chequeo de errores. En particular, nos interesa la conexión TCP dado que las conexiones realizadas por los navegadores Web con el protocolo HTTP deben usar TCP como medio de transporte, ya que esto es lo definido en la RFC. Los sockets en TCP suelen ocupar entre 4KB y 6KB de tamaño de cabecera, lo que haría en un promedio de unas 75000 conexiones, ocupando aproximadamente entre unos 290MB y 440MB en RAM (notar que esto es solo el tamaño en memoria del header).

Limitaciones de sockets. Si nos enfocamos en los sistemas operativos debemos considerar que los sockets son, inicialmente, recursos limitados. Muchos sistemas operativos permiten una cantidad aproximada de 10000 sockets, aunque este tamaño suele ser editable por entorno o inclusive aplicación. En los sistemas operativos la limitación está dada por la cantidad máxima de File Descriptors [45] que pueden inicializarse, dado que existe un mapeo uno a uno entre los File Descriptors y los sockets.

Habiendo establecido a lo largo de la sección los requerimientos necesarios para poder implementar una *conexión Comet*, tanto sea en el cliente como en el servidor, ahora pasaremos a detallar en profundidad cómo es en efecto crear un verdadero canal Comet. Cada una de las técnicas a continuación tienen un dominio de uso, dado que alguna de ellas pueden funcionar en múltiples navegadores Web (son compatibles) mientras que otras son específicas no solo de un navegador Web, sinó hasta de versiones del mismo.

3.2. Técnica general

Dentro de las posibilidades que existen para implementar Comet sin uso de plugins en los diferentes navegadores Web, tenemos la alternativa conocida como *Forever IFrame*. En particular esta técnica es bien adoptada por los navegadores Web dado que utiliza una forma básica y aceptada por la W3C de conectar el servidor con el cliente.

Solución general. Para lograr manipulación del árbol DOM vía streaming se desarrolló lo que se denomina como Forever IFrame. Esta técnica de Comet usa un elemento DOM IFrame [46] oculto, embebido en la página, para que cargue una URL distinta donde la respuesta de streaming será dada. Una vez cargada la página destino, se realiza un llamado al servidor por la URL con streaming dentro del IFrame, de forma tal que el segundo requerimiento sea el que nunca se cierra. Mientras que los datos llegan al IFrame oculto (gracias al uso de CSS y maquetación) de la página, estos suben un nivel en el árbol DOM para luego ejecutarse en el contexto de la página Web. De no ser así, se produciría un efecto de sandbox, donde las actualizaciones no llegan a visualizarse dado que están atrapadas dentro del IFrame. Hay que tener en cuenta que esto es fácilmente solucionable con un poco de manejo de Javascript.

Si observamos las líneas genéricas de implementación de Forever IFrame veremos la parte de HTML de la forma

```
<html>
  ...
  <body> ...
    <iframe url= "http://myHost/streaming.response" style=
        "hidden"/>
  </body>
<html>
```

para ir depositando la información que proviene del servidor. La función Javascript que ejecuta el código sería como se muestra a continuación

```
var foreveriframeObject = {
  initialize: function () {
```

```
        ... //se inicializa el iframe infinito
    },
    eval: function(code){
        ... //ejecutador de codigo
    }
}
```

y por último están los chunks de datos que vienen del servidor

```
'<script>
//notar como sube al DOM document escalando por "
   window.parent" para llegar al objeto que
   ejecutara el Javascript
  window.parent.foreveriframeObject.eval("alert(\"
     Hello!\")")
  </script>'
```

De esta forma, los scripts que vienen del servidor escapan la jaula del IFrame para luego poder acceder a los objetos DOM del documento HTML.

La aproximación de Forever IFrame es bastante interesante dado que funciona en casi todos los navegadores Web porque el tag IFrame pertenece al estándar de la W3C desde hace mucho tiempo. Sin embargo, esta técnica tiene varios problemas de usabilidad cuando se refiere a interfaces de usuario, dado que los navegadores Web tienen diferentes formas de mostrar que una página se está cargando:

- El puntero del mouse es dibujado como un reloj de arena (A).
- Los throbbers [47] no paran de girar (B).
- La barra de estado muestra un texto de la forma "*Esperando...*" (C).

Como se observa en una captura del navegador Web Chrome 3.3, los tres indicadores mencionados están ocurriendo al mismo tiempo

Dado que esta técnica se basa en mostrar una página que nunca se va a terminar de cargar, las notificaciones que dispone un navegador Web pueden estar mostrándose permanentemente (y en algunos casos, todas ellas al mismo tiempo). A pesar de que estos efectos gráficos no tienen ningún tipo de impacto en la lógica del la aplicación Web, dan una semántica incorrecta a la página, dado que el contenido ya fue cargado y lo que el navegador Web espera son eventos del servidor y esos indicadores le están indicando al usuario que la página no se terminó de cargar.

Figura 3.3: Diversos elementos mostrando la incompletitud de la carga

3.3. Técnicas específicas

En esta sección veremos diferentes propuestas que permiten utilizar Comet explotando las características particulares de los diferentes navegadores Web. Esto ayuda a que se solucionen diversos problemas que ocurren en el dominio del *Forever IFrame*. En particular, si no existe una técnica que sea bien adoptada por un navegador Web, se termina optando por la general, de forma tal de realizar una conexión exitosa entre el servidor y el cliente.

3.3.1. Profundización en cada técnica

En todas las técnicas, ya sea la general o las específicas, podemos encontrar dos puntos en común. El primero y más importante es que todas las técnicas son compatibles con lo definido en la W3C, lo que permite a cualquier navegador Web relativamente reciente (90' en adelante) conectarse contra un sitio con Comet y comenzar a experimentar con el mismo sin la necesidad de instalar ningún tipo de plugin. Todas las técnicas usan metodologías acordes a cada navegador Web como puede ser frames ocultos, Ajax, ActiveX y en los casos más recientes los WebSockets, debiendo elegir la metodología que más se adecúe para cada navegador. El segundo punto en común de todas estas técnicas, es que todas necesitan de un servidor con capacidades de streaming y de manipulación Javascript en el cliente, dado que los datos en forma de chunk recibidos deben ser posteriormente procesados dentro del navegador Web. A continuación se detallará cada técnica en profundidad para entender la implementación de bajo nivel que usa Comet en cada técnica.

Server-Sent events. Esta metodología introduce un nuevo tag HTML, el event-source tag [48]. El mismo funciona de manera similar a los Forever IFrames, pero la diferencia aquí es que estos no se encuentran ocultos dentro de un

contenedor, sino que están dentro de la estructura principal de la página. A ese event-source, se le determina una dirección de streaming donde debe estar escuchando para que cuando nuevos eventos sean colocados en el canal, él los tome y los procese. Para poder lograr lo anterior, es necesario realizar una vinculación entre el event-source y una función Javascript. Ésta tendrá la lógica de tomar los datos que han sido dejados en el canal, cada vez que haya datos nuevos en el mismo. El código que implementaría lo anterior se define a continuación

```
<html>
  ...
  <body> ...
    <event-source src= "http://myHost/streaming.
        response" />
    //obtenemos el unico tag event-source
    document.getElementsByTagName("event-source")[0]
      .addEventListener("algunEvento",
        manejadorDeEvento);
  </body>
<html>
```

donde es definido el evento que escuchará al canal "algunEvento". La función Javascript que funciona como un listnener y actúa cada vez que hay datos en el canal, se define como

```
function manejadorDeEvento(event){
  alert(event.data);
}
```

y por último están los fragmentos de datos que vienen del servidor, donde se define el evento y el dato que acompaña al mismo

```
Event: algunEvento

data: Hello!
```

ActiveX e IFrame. Esta técnica es prácticamente la misma que Forever IFrame. Como se mencionó previamente, los Forever IFrames traen el problema de la carga infinita dentro del navegador Web, lo que provoca una sensación de que el sitio jamás se cargará de forma completa. Para evitar esto, usaremos un framework definido en Windows, el ActiveX [49]. ActiveX es

un framework que define componentes reusables para realizar determinadas funciones dentro de un entorno Windows, todas independientes del lenguaje elegido. De toda la funcionalidad que provee el framework, nos interesa el ActiveXObject, el cual proporciona un mecanismo formal para conectarse con procesos COM [50],el cual permite definir un objeto (en particular un HTML) que puede ser utilizado como tal. En el caso de Comet, se define un objeto de tipo HTML y dentro del mismo se insertarán los otros elementos DOM, como es el caso del Forever IFrame. Todo ésto es sólo para evitar la carga infinita de los elementos gráficos del navegor Web, ya que el IFrame se encuentra literalmente oculto dentro del objeto ActiveX. El código que implementa lo descripto, es como se muestra

```
<html>
  ...
  <body> ...
    <script>
      var comet_responses = document.createElement("
          div");
      comet_responses.setAttribute("id", "
          comet_responses");
      comet_responses.appendChild(document.
          createElement("div"));

      var connection= new ActiveXObject("htmlfile");
      connection.open();
      connection.write("<html>");
      connection.write("<script>document.domain ='" +
          document.domain + "'</script>");
      connection.write("</html>");
      connection.close();
      var container = connection.createElement("div")
          ;
      connection.appendChild(container);
      container.innerHTML = "<iframe src='http://
          myHost/streaming.response'></iframe>";
    </script>
  </body>
<html>
```

Una vez definido el objeto, el servidor solo necesita enviar la información nueva (tal como se describió con el Forever IFrame), para luego ser procesada por el cliente.

XMLHttpRequest (XHR). También conocido como XMLHTTP (Extensible

Markup Language Hypertext Transfer Protocol) es una API [3] DOM usada dentro del lenguaje Javascript implementado por los navegadores Web, para enviar y recibir información al/del servidor. La respuesta que viaja dentro del objeto XHR puede ser definida en formato de texto plano o también como un documento XML [51] [52]. Con la nueva información es posible manipular el contenido actual del documento HTML sin la necesidad de recargar una nueva página. El objeto XHR define diferentes eventos que son interesantes a la hora de usarlo, como son los casos del *onreadystatechange* y el *onload*. En particular, para implementar un mecanismo de streaming nos preocuparemos por el evento *onreadystatechange*. Este evento es disparado cada vez que cambia su estado asociado, llamado *readyState*, que indica el estado en el cual se encuentra el requerimiento hecho por el objeto XHR. Este estado puede variar entre los siguientes valores:

0 (Sin inicializar).

1 (Abierto). Se abrió la conexión con el servidor.

2 (Enviado). Se hizo el requerimiento al servidor.

3 (Interactivo). Ha llegado información parcial desde el servidor, alojada en el parámetro *responseText*.

4 (Completo). Han llegado todos los datos pedidos al servidor.

En particular, el estado utilizado para esta técnica es el llamado "interactivo", ya que es invocado cada vez que llega un chunk de información desde el servidor.

Para poder utilizar este objeto, debemos escribir un fragmento de código Javascript en el sitio como se ve a continuación

```
<html>
  ...
  <body> ...
    <script>
      var httpRequest = new XMLHttpRequest();
      httpRequest.onreadystatechange = handler;
      httpRequest.open("GET", "http://myHost/
          streaming.response");
      httpRequest.send();
    </script>
  </body>
<html>
```

[3] API (Application Programming Interface) es un conjunto de funciones/métodos, que otorgan determinadas bibliotecas/objetos para ser utilizado luego por otro software de forma más abstracta.

y la función para manejar los datos se define como sigue

```
<script>
function handler() {
    if(this.readyState == 3) {
        if(this.responseXML != null && this.
            responseXML.getElementById('response').
            firstChild.data){
             // Exito, se trabaja con el dato
             ...
        }
    }
}
</script>
```

donde si el `responseXML` existe en la respuesta y además hay datos en el elemento cuya ID es *response*, se procesa la información.

WebSockets. Los WebSockets fueron concebidos para normalizar la forma de implementar Comet, y que los mecanismos realizados fueran desechados por uno estandarizado. En particular, cuando diseñaron WebSockets se enfocaron en cómo resolver diferentes problemas en torno a las limitaciones de sockets en los navegadores Web, los filtros que existen gracias a los Firewalls [53] y a los Web Proxies [54] que se encuentran en la red. También, se buscó reducir la sobrecarga de datos entre el servidor y cliente, minimizando los datos enviados entre estos para no desperdiciar ni tiempo ni espacio. Otra ventaja que aporta, es que para enviar información desde el navegador Web al servidor no usa una conexión extra con Ajax, usa la misma conexión que realizó el WebSocket tanto para enviar como para recibir datos.

WebSockets se encuentra actualmente en desarrollo dentro de HTML 5, el cual implementa de forma nativa este nuevo mecanismo para que sea soportado por todos los navegadores Web. Cuando eso ocurra, WebSockets desplazará las técnicas descritas anteriormente ya que por diseño es más eficiente: evita problemas de Firewalls, usa un único canal para enviar y recibir datos y por sobre todas las otras cosas, sera la forma estandarizada.

Para utilizar WebSockets es necesario tener dos cosas, un servidor y un cliente que lo soporte. Para poder conectarse desde un cliente con WebSockets al servidor, en el primer GET al sitio se debe definir dos campos dentro del requerimiento HTTP. El código necesario para realizar un llamado con WebSockets dentro de la página, se realiza como se muestra a continuación

```
<script>
if (``WebSocket" in window) {
```

```
    var ws = new WebSocket("ws://myHost/streaming.
        response");
    ws.onopen = function() {
      // Se conecta el websocket. Puedes enviar
         datos con el metodo send()
      ws.send("mensaje a enviar"); ....
    };
    ws.onmessage = function (evt) { var
        received_msg = evt.data; ...
      };
    ws.onclose = function() { // se cierra el
        websocket.
      };
  } else {
    // el navegador no soporta WebSocket.
  }
  </script>
```

3.3.2. Dominios de cada técnica

Dado que Comet todavía no es un estándar (al menos, si nos referimos a lo que W3C define) diferentes navegadores Web requieren diferentes técnicas para poder maximizar la experiencia del usuario. En particular, encontramos que los siguientes enfoques funcionan bien en su respectivo navegador Web:

- *Opera.* Usa *Server-Sent events*, el cual permite mandar eventos desde el servidor al navegador Web. Estos eventos son manejados por el navegador Web a través de Javascript.
- *Internet Explorer.* Usa una combinación entre *ActiveX e IFrame*. El objeto ActiveX del Internet Explorer crea una página en memoria, la cual contendrá el IFrame que carga la página de streaming. Con esta técnica, tanto la barra de estado, como el throbber no son mostrados. Lamentablemente no puede usar la técnica *XMLHttpRequest*, a pesar de tener un objeto compatible, debido a que en el estado "interactivo" no se permite el acceso a la información de la respuesta por no estar completa [55].
- *Navegadores basados en Mozilla.* El navegador Web crea una conexión *XMLHttpRequest* para luego usar el estado interactivo del objeto XMLHttpRequest parseando luego los datos que llegan desde el servidor.
- *Navegadores basados en Chromium.* Estos navegadores pueden conectarse al servidor usando *WebSockets*. En este grupo también podrían entrar futuras versiones de Safari browser y navegadores basados en Mozilla, que ya tienen en sus milestones el soporte para WebSockets.

- *Otros navegadores.* Todos los navegadores (Safari, versiones viejas de Internet Explorer, Konkeror, etc.) que no fueron contemplados anteriormente deberán utilizar la técnica de *Forever IFrame* ya que no poseen una técnica específica aún, a pesar de las desventajas nombradas previamente.

De todas las técnicas comentadas, no hay una que sobresalga por sobre las otras ya que cada una de ellas es óptima en diferentes contextos. La diferencia fundamental entre cada una de ellas, es que resuelven de manera distinta los problemas visuales que se encuentran a la hora de generar una respuesta infinita. En la siguiente Figura 3.4 se ilustra la lista comentada anteriormente

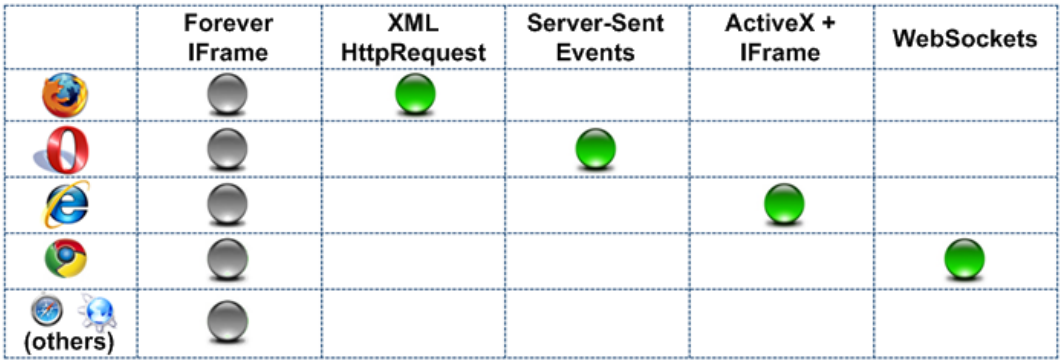

Figura 3.4: Técnicas para implementar Comet en diversos navegadores Web

las esferas verdes representan que el navegador Web soporta perfectamente la técnica, mientras que las grises indican que a pesar de realizar una *conexión Comet* éxitosa, tendrá problemas relacionados con los elementos de carga (más detalles en la Sección 3.2).

3.4. Conclusión

A lo largo de este capítulo se describieron diferentes técnicas para poder implementar la funcionalidad Comet, presentando varias alternativas que se adaptan mejor a determinados navegadores Web. Algunas involucran mayor cantidad de manipulación de Javascript, y en realidad dejan de ser un aspecto implementativo para en realidad pasar a ser “alternativas"que resuelven diferentes temas visuales presentados por Comet.

A pesar de que existe una técnica cross-brower (funcional entre todos los navegadores Web) como es el caso del Forever IFrame, se opta usualmente por generar un framework Javascript que dado el navegador Web, selecciona la mejor técnica/técnica óptima.

Por último se detalló la forma explícita para poder utilizar las diferentes técnicas de conexión explicando al final que la solución definitiva se encuentra en los

WebSockets. Estos se presentan en un estado de desarrollo, pero que a pesar de ello, muestra mejoras significativas en lo que refiere a:

- Requerimientos en el servidor y navegadores Web reducidos.
- La ausencia del problema de limitaciones de sockets en el cliente.
- Un único canal de datos para enviar y recibir información desde y hacia el servidor.
- El proceso hacia un estándar.
- Al utilizar el protocolo "ws" específicamene diseñado para los WebSockets, no existen los problemas con los Firewalls ni tampoco con los Web Proxies que puedan existir entre cliente y servidor.

Capítulo 4

Trabajos relacionados

Desde hace varios años se hace cada vez más necesario poder tener una *conexión Comet* en las aplicaciones Web para proveer un mejor servicio. Con el pasar del tiempo se fueron desarrollando diferentes aproximaciones en diversos lenguajes para proveer este servicio, como se vio en la Sección 2.1. Algunos eran muy sencillos, otros más complejos, pero en general siempre proveen esta comunicación bidireccional mediante hacks.

A continuación se verán diferentes frameworks, describiendo sus ventajas y desventajas, como puede ser el tipo de conexión que crean, el nivel de abstracción que un desarrollador posee a la hora de utilizarlo, y demás.

4.1. APE

APE [56] es un framework Open Source [57] diseñado para realizar aplicaciones Comet. El proyecto incluye un servidor Web y un framework escrito en Javascript. APE permite transferir cualquier tipo de datos en tiempo real entre un servidor Web y un navegador Web sin ningún tipo de plugins necesarios en el cliente. El proyecto se divide en dos partes que abarcan dos áreas distintas. La primera es el servidor HTTP de streaming (Comet), llamado *APE Server*. La segunda parte, el *framework Javascript APE*, que recibe los datos en el lado del cliente y los procesa siguiendo el *protocolo* definido por APE. A continuación se describirán las componentes principales de APE:

Servidor APE. El servidor APE es el encargado de manejar las conexiones Comet de los clientes. Fue desarrollado en C para que sea rápido y de poco consumo de recursos, permitiendo grandes cantidades usuarios conectados.

El servidor funciona utilizando el modelo Publisher/Subscriber [58], el cual se basa en un sistema de canales de comunicación que permiten transmitir

datos a un grupo de usuarios o a algún usuario específico directamente.

Framework Javascript. El framework Javascript provee una capa de abstracción en el cliente, para el procesado de información, recibida desde el servidor APE, proveyendo una API para que el desarollador no deba ocuparse de la comunicación explícita con el servidor.A su vez esta capa se encarga de elegir la mejor técnica de conexión dependiendo del navegador utilizado, mejorando algunos problemas que fueron vistos en la Sección 3.2.

Protocolo APE. El protocolo de APE define la forma por la cual se envía información entre el sevidor y el cliente. El protocolo, realizado con JSON [59], permite enviar y recibir varios comandos (desde el cliente al servidor) y respuestas (desde el servidor al cliente) en una única solicitud, lo que permite reducir el número de conexiones entre estos. El protocolo APE se basa en el modelo Publisher/Subscriber, lo que da la posibilidad de enviar información a más de un interesado.

Ventajas:

- Este framework es Open Source, y por este motivo, toda la comunidad Open Source puede realizar sus aportes mejorando el proyecto.
- El servidor es liviano y provee la capacidad de manejar mucha cantidad de usuarios.
- Permite elegir la técnica de conexión que más se adecue a la necesidad de la aplicación.
- No necesita el uso de plugins en el cliente ya que usa Javascript para actualizar los datos que llegan.

Desventajas:

- A pesar que APE creó una librería Javascript (JSF) para facilitar la implementación del lado del cliente, sigue siendo necesario la programación en Javascript para procesar la información recibida desde el servidor.
- No provee ninguna facilidad para tener un conjunto de tags HTML se actualicen de manera automática.

4.2. ICEfaces

ICEfaces es un framework Open Source basado en Ajax para crear aplicaciones Comet utilizando Java. Fue construido sobre otro framework llamado JavaServer

Faces (JSF) [60], el cual permite crear aplicaciones Web simplificando el desarrollo de interfaces de usuario. ICEfaces complementa este framework proporcionando una suite de componentes Ajax que facilitan el desarrollo aplicaciones Web que necesitan una *conexión Comet*.

El framework se encarga de manejar algunas complejidades de bajo nivel del mecanismo Comet, proveyendo al desarrollador con una API a nivel del servidor (Java). En realidad, el mecanismo utilizado en este framework no es de los denominados Comet si no que utiliza *Long polling*, la cual es una técnica *Client Pull* y, como ya se vio en la Sección 2.4.1, no es una de las mejores técnicas para simular una *conexión Comet*.

Basado en esta conexión simulada de Comet, provee la posibilidad de monitorear las conexiones *Long polling* que se están manteniendo con el fin de notificar cuando se detectan problemas de conexión.

ICEfaces esta compuesto por un conjunto de compoenentes que se describen a continuación:

- **Administrador de renderizado.** El Administrador de renderizado es una aplicación que coordina todas las peticiones de renderizado. Además, se encarga del registro y mantenimiento de los grupos de renderizado.

- **Grupo de renderizado.** Es un objeto que está registrado con el Administrador de renderizado, y se utiliza para organizar grupos de renderizables (clientes) que recibirán las mismas actualizaciones desde el servidor.

- **Renderizable.** Es cualquier solicitud que implementa la interfaz Renderable de ICEfaces, y se puede añadir a un grupo de renderizado, y recibir eventos desde el servidor.

Ventajas:

- Es Open Source, lo cual como se comentó en otros frameworks, es muy bueno ya que permite el uso y colaboración de toda la comunidad.

- Tiene un nivel de abstracción mayor que la mayoría de los frameworks Comet ya que permite tener fragmentos HTML que se actualizarán sin la necesidad de escribir en lenguaje Javascript.

- No necesita el uso de plugins en el cliente.

Desventajas:

- Utiliza la técnica de *Long polling*, la cual posee las deventajas vistas en la Sección 2.4.1

4.3. LightStreamer

Lightstreamer [61] es un framework Comet que permite el envío de información a través de conexiones HTTP, siguiendo el modelo Publisher/Subscriber. Este framework esta pensado para ser utilizado tanto por clientes tradicionales (aplicaciones de escritorio) como por navegadores Web, sin la necesidad de plugins (Applets, ActiveX, Flash, etc). A su vez provee una API Javascript para abstraer al desarrollador del procesado de la información enviada por el servidor y proveer una compatibilidad con muchos navegadores.

El servidor de Lightstreamer tiene varias características importantes:

- **Control de ancho de banda y frecuencia.** Cada cliente puede suscribirse a varios eventos, los cuales pueden tener una frecuencia de actualización cambiantes y provocar un tráfico en la red impredecible. Para solucionar este problema Lightstreamer posee un algoritmo de filtrado dinámico para limitar el ancho de banda, manteniendo la coherencia de los datos en general. Es posible asignar un ancho de banda máximo a cada canal de transmisión para mantener el ancho de banda utilizado.

- **Streaming adaptativo.** Varios mecanismos de adaptación son empleados por Lightstreamer para limitar el flujo de datos basado en el estado de la red. El servidor es capaz de detectar la posible congestión de la red y en estos casos cambiar la velocidad de transmisión de datos de tal manera que nunca se envían más datos de la red es capaz de manejar en un momento dado.

- **Escalabilidad.** Algunos servidores Web se han ampliado para actuar de forma streaming, pero su arquitectura tradicional esta basada en "un hilo por conexión", por lo que resulta complicado atender grandes cantidades de requerimientos. Por este motivo Lightstreamer se basa en otra arquitectura, orientada a eventos y Entrada/Salida asíncrona. La arquitectura del servidor Lightstreamer puede soportar grandes cantidades de conexiones simultáneas.

El servidor Lightstreamer está implementado en Java pero permite recibir información desde cualquier tipo de aplicacion desarrollada en otros lenguajes comunicandose mediante adaptadores.

Ventajas:

- Elige la mejor técnica de conexión dependiendo del navegador, proveyendo una buena usabilidad.

- Posee un control de ancho de banda y frecuencia que permite mantener la red sin saturar, dando un entorno equitativo para todas las aplicaciones.

- Lightstreamer fue creado con una arquitectura diferente a la de la mayoría de los servidores Web que permite mantener un gran número de conexiones simultáneas sin saturar el servidor.
- Al usar Javascript no necesita la instalación de plugins en el cliente para procesar la información.

Desventajas:

- Es pago y privativo, lo cual limita su uso y el aprendizaje de la sociedad.
- A pesar de poseer una API Javascript que facilita las actualizaciones del lado del cliente, tiene un nivel de abstracción bajo debido a que no posee ninguna posibilidad de actualizar de forma automática tags HTML.

4.4. Meteor

Meteor [62] es un framework Open Source escrito en Perl, diseñado para proveer una *conexión Comet* para enviar información desde el sevidor en tiempo real. Cuenta con el servidor Meteor, encargado de mantener las conexiones con los clientes y una clase Javascript, utilizada en el cliente para proporcionar una capa de abstracción para la recepción de datos.

Meteor consta de un servidor dividido en dos partes. En un puerto escucha por eventos de control, mientras que el segundo es para los suscriptores de los canales. Los suscriptores son clientes que se conectan en puertos y utilizan el protocolo HTTP estándar para solicitar una suscripción a un canal específico o a varios canales. Al momento de suscribirse, el cliente puede elegir el modo de interacción que desea. Los clientes se conectan en el puerto de control y utilizan un protocolo de comandos Meteor para enviar información en algún canal en particular o para realizar consultas.

Luego Meteor envía los eventos proporcionados por los puertos de control a los suscriptores del canal. De esta manera los eventos son los encargados de hacer la mayoría del trabajo duro y la generación de datos, mientras que Meteor se encarga de la tarea de entregar los datos a los suscriptores.

Los suscriptores se conectan a Meteor y hacen requerimientos HTTP estándar. Estos requerimientos pueden ser en dos formas: las solicitudes de página estática y subscripciones a canales de datos.

Ventajas:

- Es Open Source.
- Provee diferentes técnicas de conexión dependiendo del navegador utilizado, lo que permite una mejor usabilidad al usar la mejor técnica de conexión.

- No requiere la instalación de plugins en el cliente ya que procesa la información que llega desde Meteor utilizando Javascript.

Desventajas:

- Las actualizaciones en los clientes son realizadas a bajo nivel ya que debe codificarse una función Javascript que tome los datos enviados por Meteor, formatearlos y actualizar lo necesario en la página.

- Existen más de un lenguaje involucrado en la creación de un sitio utilizando Meteor, debido a que se utiliza Javascript para las actualizaciones y el lenguaje que uno elija para crear el modelo.

4.5. Orbited

Orbited [63] es un servicio HTTP que es utilizado para proveer conexiones Comet, permitiendo escribir aplicaciones Web acuatizables desde el servidor utilizando Javascript, sin necesidad de plugins externos como Flash o Applets de Java. Fue diseñado para integrarse fácilmente con aplicaciones nuevas y existentes. Una típica aplicación en Orbited se puede separar en estas 4 partes:

- **Navegador.** El navegador se conecta con el servidor Orbited a través de la librería Javascript proporcionada por el framework. Esta librería utiliza diferentes técnicas, dependiendo del navegador, para mantener una conexión de streaming.

- **Orbited.** Orbited es un servicio escrito en Python [64] que se ejecuta en el servidor. Acepta las peticiones HTTP provenientes del navegador (enviadas a través de la biblioteca Javascript que se comentó anteriormente). Orbited despacha estas peticiones a una cola de mensajes especificada por el cliente. Cuando el servidor deposita información en la cola de mensajes, se envía esta información al cliente.

- **Cola de mensajes.** La cola de mensajes es otro servicio que se ejecuta en el servidor. Los clientes pueden conectarse a ella y enviar mensajes al canal o suscribirse a un canal y por lo tanto al recibir mensajes cuando otro cliente envía otro mensaje a ese canal. Para esta cola de mensajes existen varias implementaciones, Orbited tiene el suyo pero también permite usar otros como RabbitMQ [65] o ActiveMQ [66].

- **Modelo.** Debido a que la comunicación entre el modelo y el servidor se realiza por cola de mensajes, el modelo puede ser implementado en el lenguaje que uno prefiera. Para comunicarse con el servidor, el modelo debe

conectarse a la cola de mensajes, y enviar un mensaje a un canal específico. Por este motivo, es necesario tener implementado un cliente de cola de mensajes en el lenguaje que se decida utilizar.

Ventajas:

- Es Open Source.
- Debido a su arquitectura permite integrar diferentes aplicaciones implementadas en distintos lenguajes con relativa facilidad.
- Al comunicarse con el servidor mediante una cola de mensajes, permite que el servidor escale sin demasiados problemas independientemente de Orbited.
- Al usar solo Javascript se mantiene dentro de los estándares de la Web y no es necesario el uso de plugins o instalaciones del lado del cliente.

Desventajas:

- Es necesario tener conocimiento de diferentes lenguajes para realizar una aplicación con Orbited, ya que se necesita saber un lenguaje para el modelo, Javascript para escribir las actualizaciones en el cliente, y dependiendo de la implementación de la cola de mensajes se podría necesitar otro lenguaje más.
- Existe poco nivel de abstracción en cuanto al manejo de las actualizaciones, ya que deben ser tratadas vía Javascript cuando se implementa el cliente, y se debe enviar a una cola de mensajes específica cuando se implementa el modelo en el servidor.

4.6. Pushlets

Pushlets [67] es un framework para Java orientado a Servlets [68] en donde la información es enviada desde el servidor a una página HTML sin necesidad que el cliente use Applets o plugins. A través de un único Servlet (el Pushlet), los clientes se subscriben a los eventos que deseen recibir. Cuando ocurre un evento en el servidor, los clientes que están subscritos serán notificados. Estos eventos pueden ser enviados tanto como Javascript, XML u objetos Java serializados. Este framework utiliza la técnica de *Forever IFrame* que es una de las técnicas HTTP streaming (explicadas en la Sección 3.2). La idea básica de Pushlets es enviar Javascript mediante un una conexión abierta desde un Servlet o una página JSP [68], las cuales serán interpretadas por el navegador.

El framework esta compuesto por:

- Un servidor Pushlet desarrollado en Java.
- Una librería Javascript y una página HTML para recibir eventos en páginas HTML.
- Un conjunto de clases Java para recibir eventos en el cliente, en caso de tener un cliente Java.
- Un conjunto de librerías Javascript para abstraer el procesamiento de la información llegada desde el servidor.

Las clases principales son el Servlet Pushlet, la clase Publisher, la interfaz Subscriber, y la clase Event. Al invocar al Servlet Pushlet a través de una petición HTTP, los clientes se suscriben para recibir eventos. En la solicitud se indica:

- El "asunto" por el cual el cliente esta interesado y quisiera recibir eventos.
- El formato en el cual el cliente desea recibir los eventos, ya sea Javascript, XML u objetos Java serializados.
- El protocolo utilizado.

Ventajas:

- A diferencia de otras implementaciones que usan protocolos que no son estándares, Pushlets usa los puertos y protocolos estándares de HTTP.
- No hay necesidad de tener un servidor extra. Pushlets funciona sobre cualquier servidor con soporte de Servlets, utilizando su funcionalidad (manejo de conexiones y multithreading).

Desventajas:

- Es un framework cross-browser debido a la técnica que utiliza (Forever IFrame), pero no tiene librerías que provean el mejor funcionamiento en todos los browser y todas las plataformas.
- La escalabilidad es un grave problema. Se ha probado con cientos de clientes y posee problemas con recursos, como los hilos y sockets. Una posible solución que ellos proponen sería tener un servidor dedicado que sea optimizado [69].
- Otro problema similar es que los servidores Web suelen no estar pensados para mantener conexiones vivas, por lo cual debería implementarse una solución parecida al punto anterior.

4.7. Conclusión

En este capítulo se pudo apreciar diferentes implementaciones de frameworks Comet, detallando el uso, comentando ventajas y desventajas en cada caso. Como se puede observar en el Cuadro 4.1, existen varias similitudes entre las implementaciones, así como diferencias.

	APE	IceFaces	LightStreamer	Meteor	Orbited	Pushlet
Open Source	SI	SI	NO	SI	SI	SI
Necesita otro Servidor Web	SI	NO	SI	SI	SI	NO
Provee una API de Javascript	SI	SI	SI	SI	SI	SI
Usa Streaming	SI	NO	SI	SI	SI	SI
Elige la mejor técnica de conexión	SI	NO	SI	SI	SI	NO
Provee widgets actualizables	NO	SI	NO	NO	NO	NO
Se conecta al modelo usando	Canales	Protocolo propio	Canales	Canales	Cola de mensajes	Canales
Necesita plugin en el cliente	NO	NO	NO	NO	NO	NO

Cuadro 4.1: Comparación entre frameworks

Entre las diferencias que existen en los frameworks explicados en este capítulo, cabe destacar la posibilidad de tener tags HTML que pueden ser actualizados automáticamente (provista por IceFaces), en comparación con el resto de las implementaciones que no permiten tener ese nivel de abstracción (debe programarlo el desarrollador utilizando Javascript y la API provista por cada framework). Así como se destaca el nivel de abstracción de IceFaces, también se debe comentar que es el único que utiliza la técnica de Long Polling, la cual no se considera una buena técnica en comparación con las de tipo Server Push. En la mayoría de los casos, no hay relación entre el framework que provee el servicio de Comet con el Servidor Web que aloja las páginas y el modelo de dominio de la aplicación, y por este motivo todas las notificaciones deben hacerse utilizando algún mecanismo de canales o cola de mensajes, lo cual agrega una complejidad extra. Una característica interesante entre todos los frameworks comentados en este capítulo es que proveen el servicio sin la necesidad de instalar plugins en los clientes, permitiendo el acceso a la mayoría de los navegadores Web, no solo los más actuales sino también los más antiguos. Otra característica importante, con respecto al servicio provisto por la mayoría, es la elección de la mejor técnica de conexión dependiendo del navegador a utilizar. Esto permite ofrecerle la mejor forma de tener una *conexión Comet*, aprovechando recursos y mejorando la experiencia del usuario.

En el próximo capítulo se describirá el framework propuesto en esta tesis, que trata de tomar las mejores características de todos los frameworks investigados e implementarlas para que sean simples de utilizar. Entre ellas se encuentran el uso de Streaming, la elección de la técnica óptima para cada navegador, el uso de estándares Web, la abstracción en el uso de Javascript y, lo más importante a nuestro entender, proveer un nivel de abstracción que permita la posibilidad de utilizar widgets (tags HTML) que se actualicen automáticamente cuando sucede

algún evento en algún aspecto del modelo del cual están interesados.

Capítulo 5

Meteoroid

Este capítulo se focalizará en explicar el framework **Meteoroid** en detalle, comenzando por una aproximación general, profundizándose luego en las diferentes capas. Terminamos con un apartado de los diversos problemas que nos encontramos, para finalizar con ejemplos y comparativas con otros frameworks Comet.

5.1. Introducción

Meteoroid es un framework para el desarrollo de aplicaciones Web *cometificadas*, centrado en otorgar un nivel de abstracción lo suficientemente elevado para permitir el desarrollo de aplicaciones con Comet de una forma rápida y simple. Este framework fue desarrollado en VisualWorks [5], funcionando sobre el servidor Web Swazoo y como un agregado del framework Seaside [2].

Seaside es un framework que permite el desarrollo de aplicaciones Web en Smalltalk. Combina un modelo orientado a objetos con "Continuations" [70], permitiendo múltiples flujos de control para una misma página. Este framework toma los fragmentos de HTML como componentes, los cuales pueden ser reusados en diversas aplicaciones. Una de las ventajas más grandes de Seaside es que las aplicaciones son escritas en Smalltalk, evitando el uso de HTML y permitiendo al desarrollador abstraerse de las etiquetas HTML, utilizando el mismo lenguaje con el que desarrolló el modelo. Por ejemplo, para renderizar el siguiente código HTML:

```
<html>
    <body>
        <h1>Ejemplo</h1>
        <div id='ejemploID'>
```

```
            Hola mundo :)
        </div>
    </body>
</html>
```

debo crear una clase, llamémosla `HolaMundo`, la cual heredará de `WAComponent` y tendrá reimplementado el mensaje `#renderContentOn:` de la siguiente manera:

```
HolaMundo>>renderContentOn: html
    html heading level: 1;
        with: 'Ejemplo'.
    html div id: 'ejemploID';
        with: 'Hola mundo :)'.
```

Como se ve es muy sencillo crear aplicaciones Web utilizando dicho framework. Para tener una mayor información sobre los beneficios y sobre el uso de este framework mirar el Apéndice A.

Meteoroid esta compuesto por tres capas, cada una desarrollada a partir de las anteriores, proveyendo mayor funcionalidad y abstracción. Estas mejoras se realizan con respecto a la *conexión Comet* creada para este tipo de aplicaciones, como a la facilidad de actualización entre el modelo y las vistas Web.

Este framework fue desarrollado con varios objetivos. Por un lado, proveer una *conexión Comet* para las aplicaciones desarrolladas en Seaside, permitiendo crear aplicaciones interactivas que reciban información desde el servidor en tiempo real. Además de agregar esta funcionalidad, es importante que dicha conexión sea creada de forma óptima dependiendo cada navegador Web en donde se este utilizando. Por otro lado, dentro de los objetivos se encuentra proveer una metodología de desarrollo que permita crear aplicaciones Web de forma fácil y simple, abstrayéndose tanto de las conexiones como de las actualizaciones a bajo nivel, las cuales suelen estar hechas utilizando Javascript. Con respecto a este último punto, consideramos que la metodología utilizada por VisualWorks para el desarrollo de aplicaciones de escritorio es muy interesante (ver Apéndice B) ya que provee una forma simple pero muy efectiva de crearlas. Por este motivo, decidimos utilizar esa metodología en el desarrollo de aplicaciones Web "vivas".

En las próximas secciones se explicará en detalle la arquitectura del framework de forma exhaustiva, detallando los usos de cada capa, comentando los pros y contras de cada una, así como también describiendo ejemplos de su uso.

5.2. Arquitectura

En el transcurso de esta sección se desarrollará una explicación de la arquitectura de **Meteoroid**, comenzando por una descripción general para comprender

su funcionamiento, para luego centrarse en cada capa, explicando su implementación.

5.2.1. General

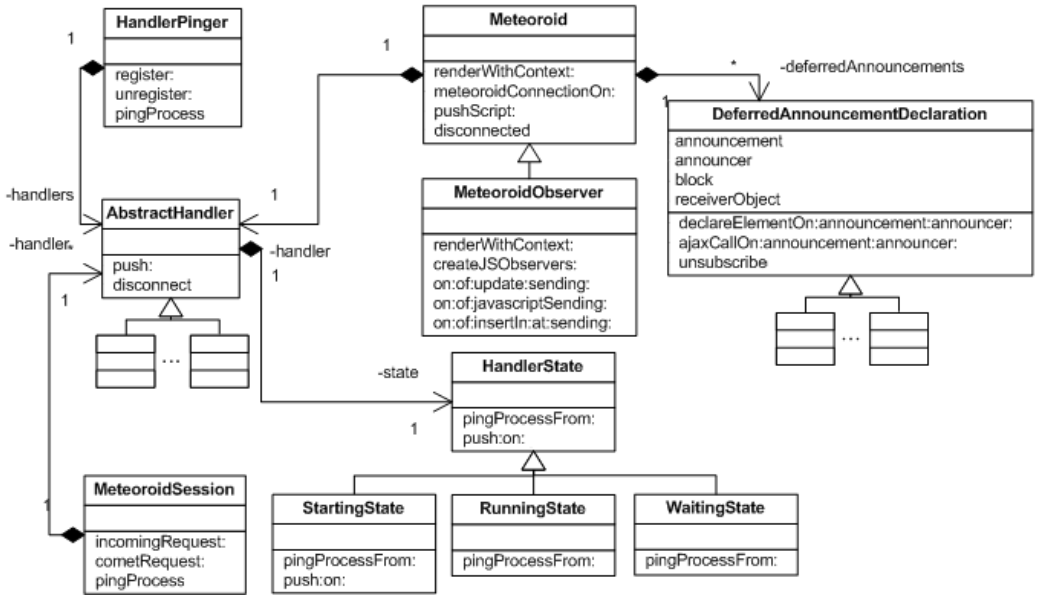

Figura 5.1: Diagrama de clases de Meteoroid

Meteoroid está compuesto por un conjunto de clases diseñadas en torno a la clase `Meteoroid`, la cual permite continuar desarrollando aplicaciones de la misma forma que se hacían utilizando Seaside, pero contando además con la funcionalidad extra de poder enviar información desde el servidor al cliente cuando ocurra algún evento en el modelo.

Una de las características importantes que se tuvo en cuenta para la creación del framework fue que la *conexión Comet* provea un buen servicio a los clientes que utilicen la aplicación Web. Es por esto que **Meteoroid** se encarga de elegir la manera óptima para crear la *conexión Comet*, dependiendo del navegador Web que el cliente esté utilizando, teniendo en cuenta lo explicado en el Capítulo 3.

Como se ha comentado en secciones previas, es importante que la *conexión Comet* provista sea simple y fácil de utilizar. Por este motivo, se realizaron una serie de modificaciones y extensiones a Seaside logrando que agregar **Meteoroid** a una aplicación Seaside sea tan fácil como seguir dos pasos:

- Heredar de la clase `Meteoroid` en vez de la clase `WAComponent`.
- Cambiar la sesión por defecto que utiliza Seaside (`WASession`) por una sesión provista por el framework llamada `MeteoroidSession`.

Una vez realizado estos pasos contaremos con una *conexión Comet* y un protocolo que permite la utilización de dicha conexión.

Meteoroid consta de tres capas (ver Figura 5.2), creadas una encima de la otra, las cuales agregan funcionalidad y proveen un mayor nivel de abstracción a medida que se eleva de capa. La capa más baja del framework es la llamada **PushScript**, la cual es la capa base del framework y es la encargada de proveer una *conexión Comet* junto con un mensaje que permite el envío de código Javascript a la página que se encuentra en el cliente para poder manipular la información.

Como sabíamos que era posible mejorar este protocolo mejorando la forma de manipular el navegador Web, se creó una nueva capa llamada **Observer**, que agrega una forma de automatizar actualizaciones en el momento de la creación de la página y permite dibujar HTML con código Smalltalk. Como se explicará más adelante, la capa **Observer** provee un nivel mucho mayor de abstracción.

Pero como la idea del framework es llegar a implementar aplicaciones Web de manera similar a como se programa en aplicaciones de escritorio (ver Apéndice B), **Meteoroid** posee una capa más, llamada **Web Live Widgets**, que otorga el máximo nivel de abstracción mediante widgets que son actualizados automáticamente dependiendo de algún aspecto del modelo. En las próximas secciones se detallarán las diferentes capas provistas por **Meteoroid** de forma ascendente, explicando las características de cada una y mostrando la facilidad de uso mediante ejemplos.

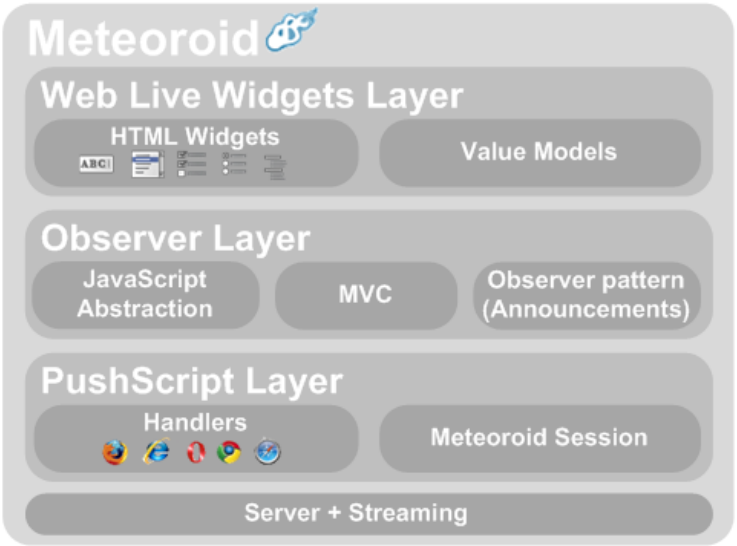

Figura 5.2: Separación en capas de Meteoroid

5.2.2. PushScript

La capa PushScript es la primer capa del framework Meteoroid, encargada en trabajar a bajo nivel, inicializando y manteniendo la *conexión Comet*, proveyendo un protocolo por el cual se envía Javascript desde el servidor al cliente. Además de proveer este protocolo, `PushScript` es el núcleo de **Meteoroid**, por lo que las capas siguientes (**Observer** y **Web Live Widgets**) se construyen encima de éste. A pesar de su bajo nivel para poder manipular la vista del cliente, esta capa destaca su simplicidad y la forma directa de editar el árbol DOM del navegador Web vía Javascript.

Funcionalidad provista

La clase `Meteoroid` provee al desarrollador el mensaje `#pushScript:`, el cual recibe como parámetro un fragmento Javascript que se evaluará en el navegador Web que puede ser un llamado a una función, una modificación de algún objeto DOM, etc. Por ejemplo, consideremos una clase `MeteoroidExampleAlert`, la cual hereda de la clase `Meteoroid` y posee un mensaje `#showDialog` definido de la siguiente manera:

```
MeteoroidExampleAlert>>showDialog
    self pushScript:'alert("Hola");'
```

Cada vez que se llame a `#showDialog` en cada instancia de `MeteoroidExampleAlert` del servidor, se levantará una notificación en el navegador Web con el mensaje "Hola". A continuación se elaborará un ejemplo más complejo para mostrar que continúa siendo sencillo el uso de esta capa del framework.

Consideremos el ejemplo del chat que hemos venido desarrollando en otros capítulos, centrándonos en la actualización de la lista de mensajes (observar la Figura 5.3 para clarificar esta explicación). El modelo de la aplicación constará de tres clases:

- `Message` Representa al mensaje, contiene el texto y el usuario que lo escribió.
- `User` Representa a un usuario conectado al chat, agrega mensajes en la sala.
- `Room` Representa la sala de chat, contiene una colección de mensajes y de usuarios, y cada vez que alguna de las colecciones cambia es el encargado de avisarle a todos los usuarios de los cambios ocurridos.

A su vez tendremos una vista realizada con nuestro framework, y por lo tanto tendremos una clase `MetChat`, que hereda de `Meteoroid` para poder enviar las actualizaciones desde el servidor. Para mantener actualizada la página Web de una forma modular se utilizará el patrón Observer (ver Sección B.2), donde cada

usuario será el *sujeto* de la vista `MetChat`. De esta manera, cada vez que un usuario envía un mensaje a la instancia de `Room`, éste le notificará de la llegada de un nuevo mensaje a todas las instancias de `User`. Cada usuario notificará a todos sus dependientes mediante un `change: #message with: aMessage`. La instancia de `MetChat`, al ser dependiente, recibirá el mensaje `#update:with:`, el cual fue implementado de la siguiente forma para actualizar la lista de mensajes que se muestra en el navegador Web:

```
MetChat>>update: aSymbol with: aMessage
    |str|
    str := 'insertMessage(" '
        aMessage user, ' ", " '
        aMessage body,
        , ' ");' .
    self pushScript: str.
    ...
```

donde la función Javascript `insertMessage(user, message)` agregará el nuevo mensaje utilizando técnicas DOM. La función `insertMessage(..)` se definiría de la siguiente forma:

```
function insertMessage(user, message) {
    var div = document.createElement('div');
    div.innerHTML= user.bold() + ' dice:' + message
       ;
    document.getElementByID('messageList').
       firstChild.appendChild(div);
}
```

donde *messageList* es el ID del elemento contenedor de todos los mensajes del chat, *user* es 'Usuario_1' y el mensaje *message* es 'hello', el código final en HTML será:

```
<html>
   ...
  <body> ...
    <div id='messageList'>
        ...
        <div><bold>Usuario_1</bold> dice: hello</div>
    </div>
  </body>
<html>
```

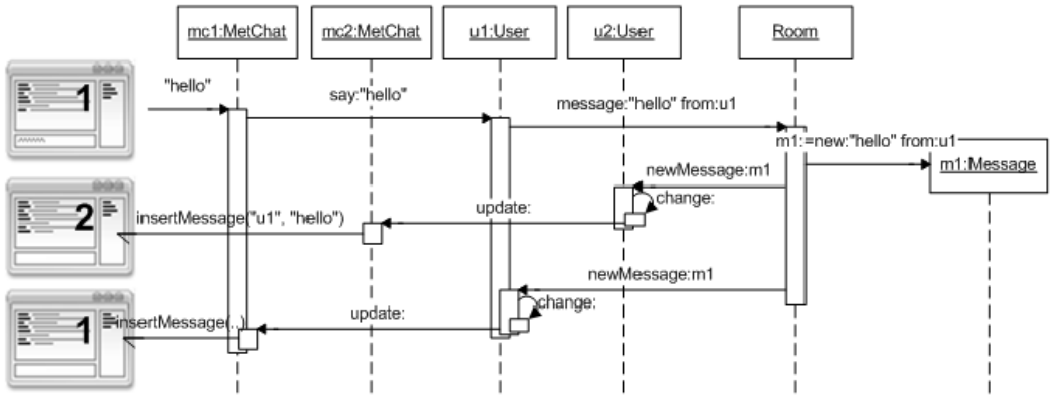

Figura 5.3: Actualización de chat usando la capa PushScript

Se verá a continuación cómo se logró la mencionada funcionalidad, explicando los tipos de conexión, las elecciones de técnicas, cómo se modela el estado de las conexiones, cómo se mantiene viva la aplicación, y por último el componente clave del framework, la clase `Meteoroid`.

Tipos de conexión

En el Capítulo 3 se habló de los distintos tipos de conexiones posibles, dependiendo de las tecnologías soportadas por los diferentes navegadores Web, entre las cuales están:

- Forever IFrame
- XMLHttpRequest
- Server-Sent events
- ActiveX + IFrame
- WebSockets

De las cinco anteriores, las cuatro primeras se encuentran contempladas dentro de **Meteoroid**. Esto es así porque nuestro framework depende directamente de Seaside y Swazoo [71], el primero es el encargado de dibujar las aplicaciones Web mientras que el segundo es un servidor HTTP implementado netamente en Smalltalk. El problema con WebSockets es que no es una técnica sino un nuevo protocolo, el cual debe ser implementado por Swazoo a nivel del servidor, pero por el momento no hay ninguna implementación y como esto escapa a este trabajo de grado, decidimos no soportarlo por el momento. Por lo tanto para navegadores basados en Chromium, se optó por la técnica de Forever IFrame.

Los handlers son el artefacto por el cual las componentes `Meteoroid` pueden romper con el protocolo HTTP y enviar información desde el servidor al navegador Web. Su rol es determinante en el framework y están estrechamente vinculados a cómo optimizar el tipo de conexión a realizar. Cada handler tiene en su estructura el stream (respuesta infinita del servidor al cliente) por donde serán enviados los datos, el estado actual de la conexión, y otros estados que se explicarán más adelante. `Meteoroid` representa las conexiones en la jerarquía de `AbstractHandler`, como se ve en la Figura 5.4.

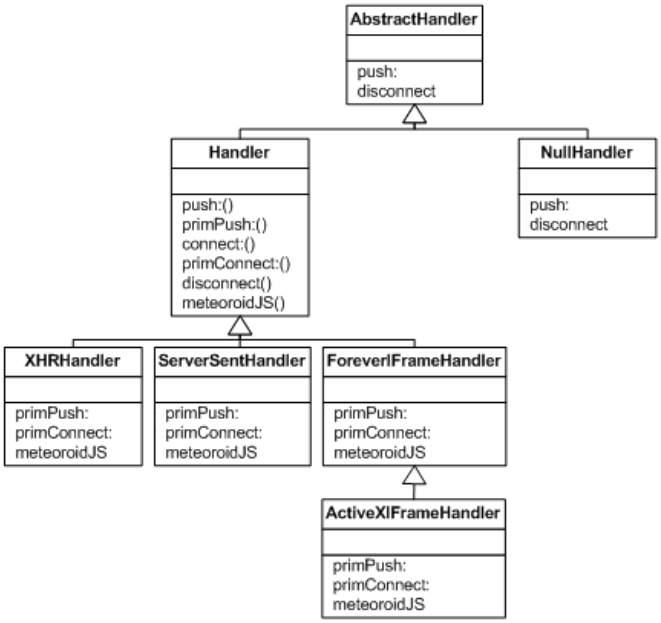

Figura 5.4: Jerarquía Handler

Es interesante remarcar que además existe la clase `NullHandler`, la cual no mapea con una técnica de conexión "real". Esta clase es útil para inicializar a las diferentes componentes `Meteoroid`, ya que una todas las intancias recién creadas poseen un handler por defecto, y tener un handler que entiende los mensajes pero "no hace nada" resulta útil en los primeros instantes. `NullHandler` entenderá el protocolo provisto por la jerarquía `AbstractHandler`, manteniendo así el comportamiento y consistencia del framework. En particular `NullHandler` es una implementación del patrón NullObject [1] [72, Null Object pattern].

[1] El propósito del NullObject es encapsular la ausencia de un objeto proveyendo una alternativa que no realiza acción alguna. Es en síntesis, un objeto que "no hace nada", pero que tampoco rompe el esquema.

Elección de la técnica de conexión

La *conexión Comet* con el servidor se realiza mediante una objeto Javascript, residente en el navegador Web, creado para inicializar la conexión una vez que la página ha sido desplegada de forma completa en el navegador Web. Esto es posible gracias al evento `document.onLoad()` del documento HTML [73, Intrinsic events], el cual ejecuta el código especificado dentro de esa función solo después que la página se termina de cargar. El llamado de nuestra función crea e inicializa un objeto Javascript llamado `meteoroid`, el cual posee un conjunto de variables que mantienen el estado de la conexión, la URL a donde mira la *conexión Comet*, etc. Además define cuatro funciones:

- `bindURL(url)`. Dada una URL, la guarda en una variable del objeto `meteoroid`. Esta URL es la que apunta al servidor donde la respuesta va a ser infinita, y por donde el servidor va a enviar actualizaciones al cliente.

- `initialize()`. Realiza dos tareas, la primera es crear un contenedor donde los contenidos nuevos del servidor serán interpretados y la segunda es crear la conexión contra el servidor. Lo interesante es que esta función es distinta en cada tipo de conexión y es por esto que los diferentes handlers la reimplementan, definiendo lo que sea acorde a cada técnica.

- `onUnload()`. Cada vez que la página se cierre, ya sea porque se cerró la pestaña, el navegador Web por completo, o se haya navegado hacia otra página, el evento `onUnload()` va a disparar un llamado al servidor notificando que la conexión debe cerrarse.

- `eval()`. Esta función cumple el propósito de unificar el medio de enviar información a las diferentes páginas, independientemente del tipo de navegador Web que se utilice.

Una vez que la conexión ha sido creada, el servidor podrá enviar información cuando lo considere necesario. Como ya se mencionó, para elegir la conexión se debe cambiar la sesión por defecto en Seaside por `MeteoroidSession`. Esta nueva sesión es la encargada de elegir el mejor handler para la *conexión Comet* usando una instancia de la clase `HandlerFactory`, la cual elige entre la jerarquía de clases de `AbstractHandler` cual es la mejor técnica de conexión a instanciar.

El `HandlerFactory` contiene una colección de los handlers disponibles y cuando una sesión requiera iniciar una conexión le pedirá a esta clase un handler para su navegador a través del User-Agent [74]. Por lo tanto, cada subclase de `Handler` debe implementar un protocolo específico:

- `#canHandle: aUserAgent` (mensaje de clase) El cual deberá responder si puede ser utilizado en el navegador que llega como parámetro.

- `#meteoroidConnectionScript` (mensaje de instancia) Donde están definidas las tres funciones Javascript de `meteoroid`.
- `#primConnect: aRequest` (mensaje de instancia) Que sirve para configurar la conexión como streaming y agregar los headers HTML necesarios para la técnica específica.
- `#primPush:aString` (mensaje de instancia) Que codifica el chunk de respuesta en el formato requerido para esa técnica específica.

Si tomamos como ejemplo la clase `XHRHandler` la cual trabaja con navegadores basados en Mozilla, los mensajes quedan definidos de la forma:

```
XHRHandler(class)>>canHandle: aUserAgent
  ^('*Firefox*' match: aUserAgent)
    or:[ ('*Iceweasel*' match: aUserAgent)]
```

los asteriscos funcionan como un comodín y el mensaje `#match:` retorna `true` en caso de que existan ocurrencias del string *Firefox* o *Iceweasel* dentro del parámetro `aUserAgent`.

```
XHRHandler>>meteoroidConnectionScript
^'/* Meteoroid API */
var meteoroid = {
    connection  : false,
    container   : false,
    url : false,
    bindURL: function(url) {
        ...
    },
    initialize: function () {
        ...
    },
    onUnload: function() {
        ...
    }
    init: function() {
        ...
    }
}'
```

Prepara el objeto `meteoroid` para que sea luego ejecutado en el navegador Web.

```
XHRHandler>>primConnect: aRequest
```

```
self streamedResponse: aRequest nativeRequest
   streamedResponse.
self streamedResponse contentType:
  'multipart/x-mixed-replace;boundary="
    limite01234"'
```

Setea la respuesta como una de tipo streaming, y luego cambia el content-type para proporcionar información al navegador Web sobre la codificación a utilizar, que en este caso es multipart.

```
XHRHandler>>primPush: aString
  self streamedResponse nextPut: Character cr.
  self streamedResponse nextPut: Character lf.
  self streamedResponse nextPutAll: aString.
  self streamedResponse flush
```

Por último, el mensaje `#primPush:`, el cual indica de forma específica en la cual deben enviarse los datos del servidor al cliente, en el caso particular de la técnica XMLHttpRequest luego del contenido, se termina con el método `#flush` (el cual le indica al stream que debe enviar los datos al navegador Web).

Vale la pena destacar que si surge una nueva técnica sólo basta con heredar de `Handler`, definir los tres mensajes de instancia e implementar el mensaje de clase `#canHandle:` que determina cuando puede ser usado el nuevo handler. De esa forma alcanzaría para manipular una nueva técnica con el framework Meteoroid.

Estados de la conexión

Cada objeto `Handler` representa el canal abierto entre el servidor y el cliente como se mencionó anteriormente, y cada vez que una componente `Meteoroid` desea hacer un `#push:`, en realidad es su `Handler` asociado quien se encarga del envío de la información al cliente. Esta conexión puede estar en diferentes estados y por este motivo fueron representados en el framework siguiendo el patrón State [2] [75, State pattern]:

StartingState. Este estado representa la inicialización de la conexión. Durante este estado no se puede enviar información del servidor al cliente.

RunningState. Este estado es utilizado cuando el `Handler` ya ha sido inicializado y esta habilitado para enviar información al cliente. Cuando se encuentra en este estado el `Handler` mantiene la sesión viva permitiendo que la conexión se mantenga abierta. Si ocurre algún error al enviar la información,

[2] El patrón State permite a un objeto alterar su comportamiento cuando su estado interno cambia. Cada vez que cambia el estado, el objeto parecerá de una clase distinta.

el `Handler` cambiará su estado a `WaitingState`. Si un handler está en este estado entonces las componentes `Meteoroid` pueden enviar información a los navegadores Web.

WaitingState. Este estado indica que la conexión no se encuentra disponible para enviar información al navegador Web. Esto puede ocurrir por dos motivos, el primer caso es que el cliente se retire de la página, ya sea cerrando la ventana o navegando hacia otra página, pudiendo en este último caso retornar luego. El segundo caso es provocado cuando ocurre algún error y la conexión es destruida.

Mediante este diseño se permite modificar el comportamiento de las componentes de una forma flexible dependiendo del estado de la conexión.

Mantenimiento de conexiones

Para poder explicar este punto, primero debemos adentrarnos un poco en lo que es el núcleo de Seaside. Como se explica en el Apéndice A, Seaside tiene un excelente manejo de control de flujo gracias al uso de Continuations, que son básicamente procesos (comparable con stacks) las cuales pueden ser resumidas muchas veces. Gracias a esta particularidad, las Continuations se podrían mantener vivas y no ser eliminadas del sistema, llevando a un crecimiento en memoria importante ya que cada Continuation es un objeto en memoria que consume espacio y procesamiento. Es por tal motivo que Seaside manipula el tiempo límite de las mismas a través de las sesiones. Todas las aplicaciones de Seaside cuentan con una sesión, `WASession`, que se encargan de mantener viva la aplicación a través de marcas de tiempo. Una sesión expira por dos métodos distintos: se le dice explícitamente que expire (usando el método `#expire`) o cuando no se actualice la marca de tiempo. Esta marca de tiempo es modificada cada vez que el usuario de una aplicación Web realiza alguna acción (click en un link, hace submit de un formulario HTML, etc.), y se establece como nueva marca la hora actual. Si el usuario no realiza acción alguna sobre la aplicación, el tiempo límite de la sesión llega a su límite y por lo tanto la aplicación toma el comportamiento como si estuviera cerrada. Esto permite que el GarbageCollector [76] no se lleve a una sesión cuando está "viva", sino cuando esté expirada o el tiempo de vida haya caducado.

Este escenario funciona bien para las conexiones normales de Seaside, pero para las conexiones Comet no sirve debido a que pueden pasar más del tiempo establecido sin interacción y luego producirse una ráfaga de cambios que requerirán el uso de la aplicación por parte del cliente. Por este motivo se requiere de un objeto que mantenga estas sesiones abiertas cuando las páginas en los navegadores Web estén activas. En Meteoroid, el objeto encargado de esta acción es el `HandlerPinger`, un único proceso que se encuentra en el sistema mientras haya

alguna conexión activa. Cada componente `Meteoroid` en su inicialización registra su `Handler` en este objeto y lo remueve cuando la componente deja de ser usada. El `HandlerPinger` debe ser visto como un mantenedor de conexiones de Seaside, en particular existe uno solo y es compartido por todas las componentes `Meteoroid` pero es redefinible en caso que se quiera utilizar otro con distintas restricciones.

No sólo es necesario saber mantener vivas las conexiones sino saber cuando desconectarlas. Crear aplicaciones Web colaborativas para ser usadas desde un servidor tiene ventajas como la participación en grupo, no forzar a todos los integrantes a estar físicamente en el mismo espacio, trabajar todos al mismo tiempo, etc. Esta ventaja acarrea por otros lados algunas desventajas, como la disponibilidad física del servidor, la cual no depende de los usuarios pero sí los afecta. Es por este motivo que es importante que todas las aplicaciones que estén corriendo deban ser notificadas cada vez que la imagen de VisualWorks va a ser cerrada y darle la oportunidad de quedar en un estado consistente, ya sea guardando información necesaria, como borrando dependencias, etc.

Es por tal motivo que se creó la clase `HandlerSystem`, subclase de `Subsystem` [3], que define dos mensajes *hook*, `#setUp` el cual se ejecuta al inicio y `#tearDown` cuando la imagen es cerrada. Ambos métodos hacen un llamado al mensaje `#clearAllMeteoroids`.

```
HandlerSystem>>clearAllMeteoroids
    Meteoroid allSubclasses
        do: [:meteoroidClass |
                self clearMeteoroid: meteoroidClass
            ]
```

De esta forma, cada vez que se cierra la imagen todas las dependencias y asociaciones creadas por el framework son removidas y a su vez todas las componentes `Meteoroid` activas en ese momento son notificadas del cierre. Lo mismo es realizado cuando se inicia la imagen, por si existe alguna componente que haya quedado inconsistente.

El "componente": Meteoroid

Como se explica en el Apéndice A las componentes que quieran mostrarse en una página deben redefinir el método `#renderContentOn:` para que se ejecute el *hook method*. Esto es porque las componentes son en realidad una implementación

[3] `Subsystem` coordina el startup/shutdown de la imagen de VisualWorks, junto con otros eventos de notificación, opciones de línea de comando, llamando a sus subclases para ejecutar `setUp`/`tearDown` según corresponda.

del patrón Template Method [4] [75, Template Method pattern], donde sólo redefiniendo el método de renderizado alcanza para poder dibujar cada componente.

Esta forma de construir aplicaciones es muy útil y a la vez muy simple. Por tales motivos, a la hora de implementar el framework Meteoroid se tomó como objetivo no modificar el mecanismo que tiene Seaside para renderizar cada componente. Analizando, se observó un punto en común en la etapa de dibujado, el método `#renderWithContext:`. Este método realiza la tarea de inicializar el canvas a dibujar (el HTML), y verifica si es necesario introducir la aplicación en modo debugging, para luego delegar la responsabilidad de pintar la componente a través del `#renderContentOn:`. Si observamos el código Seaside:

```
renderWithContext: aRenderingContext
    | html callbacks |
    callbacks := aRenderingContext callbacksFor:
       self.
    html := self rendererClass context:
       aRenderingContext callbacks: callbacks.
    (self showHalo and: [ aRenderingContext
       isDebugMode ])
        ifTrue: [ (WAHalo for: self)
           renderContentOn: html ]
        ifFalse: [ self renderContentOn: html ].
    html flush
```

donde primero crea un objeto donde los callbacks van a ser contenidos (`callbacks`), inicializa un nuevo objeto para renderizar (`html`) y finaliza preguntando si está en modo de debug. En caso de estarlo inserta los halos (herramienta para poder debuggear aplicaciones Web a través de Seaside) y termina llamando al método hook `#renderContentOn:`, el cual debe ser redefinido en cada componente.

Todas las componentes de Seaside pasan por este proceso cada vez que se van a pintar, independientemente si son la componente `root` de una aplicación o cualquier otra. El último detalle a tener en cuenta, recordando lo que se menciona en el Apéndice A, es que las componentes pueden anidarse tantas veces como se quiera. Este característica es importante, porque todas las componentes a dibujar pasarán siempre por un `#renderWithContext:` y luego por un `#renderContentOn:`.

Para agregar Meteoroid a las aplicaciones, el `#renderWithContext:` se modificó como se muestra a continuación:

```
Meteoroid>>renderWithContext: aRenderingContext
```

[4] Este patrón permite definir el esqueleto de un algoritmo, dejando algunas operaciones ser definidas por las subclases. Permitiendo entonces que la estructura del algoritmo no cambie, sino que cada subclase lo modifique parcialmente.

```
| html callbacks meteoroidBindWasWritten |
callbacks := aRenderingContext callbacksFor:
   self.
html := self rendererClass context:
   aRenderingContext callbacks: callbacks.
(self showHalo and: [ aRenderingContext
   isDebugMode ])
    ifTrue: [ (WAHalo for: self)
       renderContentOn: html ]
    ifFalse: [
        meteoroidBindWasWritten := self
           meteoroidConnectionOn: html.
        self insertDisconnectScript: html.
        self renderContentOn: html.
        self prepareHandler:
           meteoroidBindWasWritten].
html flush
```

en sí el método es casi idéntico, salvo que se tienen ciertas salvedades antes de llamar al hook method `#renderContentOn:`. Por ejemplo, se guarda en una variable el valor booleano en `#meteoroidBindWasWritten` para saber si la componente actual fue el primero que llamó al `#renderWithContext:`, se insertan llamados Javascripts para que una vez finalizada la conexión en el navegador Web el servidor tome acción y mate la conexión, y por último reinicializa un estado del handler.

De estos tres cambios, el primero es en realidad el más importante, dado que `#meteoroidConnectionOn:` inicializa el estado global de la componente. Este mensaje tiene además la semántica de detectar si es la componente root o alguno de sus hijos, y actuar acorde la situación. En caso de ser la componente root, debe instanciar un nuevo `Handler`, y dejar las bases para que se ejecute el código pertinente en el navegador Web (en particular, insertar el llamado Javascript que inicia la *conexión Comet*). Si por el otro lado fuera un componente hijo, tener en cuenta que el handler ya está creado y que no hace falta insertar código Javascript, solo hace falta asociar el mismo handler ya creado con el componente hijo. De esta forma, tenemos la ventaja que un único handler será manipulado a lo largo de la vida de la aplicación, optimizando recursos (ver Sección 3) en el servidor (apertura de sockets), como en el cliente (limitaciones asociadas a cantidad de conexiones abiertas en paralelo).

Resumen

Como se observó en la **Funcionalidad provista**, la capa menos abstracta de **Meteoroid** provee grandes funcionalidades: elección de técnica, manipulación de

handlers, anidación múltiple y manejo sencillo del handler y un mensaje que permite enviar información desde el servidor al cliente. Este método `#pushScript:` tiene una vasta usabilidad para manipular el documento HTML a través del árbol DOM, pero es cierto que a gran escala es muy complejo manejar múltiples elementos DOM. Codificar Javascript para elementos visuales no es lo más sencillo, dado a que no existen herramientas lo suficientemente evolucionadas como para escribir código Javascript y testearlo en todos los browsers, garantizando que el comportamiento sea siempre el mismo.

Es cierto que existen librerías de alto nivel para Javascript, como Prototype, Scriptaculous, jQuery, MooTools, etc., pero también es verdad que cuando una persona desea tener 10 inputs con determinadas propiedades, no es lo más agradable tener que cambiar de un contexto de programación (y debugging) en Smalltalk a uno Javascript para verificar que todo anduvo de forma correcta. Es por esto, que Meteoroid provee más funcionalidades, que serán explicadas a continuación.

Como se vio en ambos ejemplos, es muy sencillo utilizar esta capa, debido a que el desarrollador no debe mantener el control de la conexión, sólo utiliza el protocolo provisto por el framework, quien se encarga de elegir la mejor conexión dependiendo del navegador utilizado, para luego mantenerla abierta y enviar la información que el desarrollado requiera enviar.

5.2.3. Observer

Las aplicaciones que son creadas en Smalltalk son diseñadas en un entorno orientado a objetos y cada framework que es creado en este ambiente trata de seguir este enfoque. Aunque la primer capa provee una forma simple de crear aplicaciones Comet, no mantiene este tipo de programación utilizado en todo el entorno tanto como se quisiera. Cada vez que se desee actualizar un fragmento del sitio Web, se debe escribir código Javascript, lo que agrega complejidad al mezclar a mitad de un método Smalltalk fragmentos de código Javascript. A su vez, en una aplicación compleja estas actualizaciones en las vistas Web se realizarán cuando ocurra alguna modificación en el modelo, y si se encuentra bien diseñado, estas notificaciones seguramente se producirán siguiendo el patrón Observer (como se mostró en el ejemplo del chat en la capa PushScript). Pero para implementar el patrón Observer se debe llevar un control de las dependencias entre el modelo y la vista, registrándolas cuando la aplicación es accedida y removiéndolas cuando se retiran, lo cual no es sencillo de realizar ya que se debe tener en cuenta que sucede cuando se refresca la página, cuando se hace click en los botones back/forward, etc. Los problemas comentados fueron los que motivaron la creación de una nueva capa que proveyera un nivel de abstracción mayor a la provista por la capa PushScript, donde solo es necesario saber codificar en Smalltalk y el resto (mecanismo de dependencias) corre por cuenta Meteoroid.

Funcionalidad provista

Esta capa de Meteoroid provee un protocolo por el cual el desarrollador de la aplicación Web puede definir actualizaciones sobre eventos sucedidos en el modelo de una forma simple, dejando que la componente `MeteoroidObserver` (subclase de `Meteoroid`) se encargue de crear/deshacer dependencias y enviar actualizaciones desde el servidor al cliente cuando sea necesario. De esta forma el desarrollador puede olvidarse de las actualizaciones y las dependencias, concentrándose solamente en los problemas de la aplicación en sí.

El protocolo provisto por esta capa provee una serie de mensajes que, utilizando la implementación de Announcements [77](ver Apéndice B), definen que cada vez que se notifique una instancia de alguna subclase de `Announcement` (`anAnnouncement`) desde una instancia subclase de `Announcer` (`anAnnouncer`), se haga una determinada acción. Dicha acción, posiblemente encuentre involucrado a un `WARenderCanvas` (`html`), el cual será utilizado para dibujar los nuevos elementos HTML codificando siempre en Seaside. Veamos a continuación los diferentes mensajes que la capa **Observer** implementa:
Updaters. Estos permiten actualizar diferentes elementos de una página, reemplazando su contenido por uno nuevo. El objetivo es modificar un contenido DOM que se encuantra en la página Web, cada vez que ocurre algún cambio en el modelo, y para lograrlo los *updaters* suelen utilizar el `html` para escribir el nuevo contenido en el HTML.

- **on: anAnnouncement of: anAnnouncer update: anId callback: aBlock** Actualiza el elemento DOM cuyo id sea `anId` usando el bloque de renderizado `aBlock`. En este caso el bloque puede recibir cero, uno (`html`), dos (`html`, `anAnnouncement`) o tres (`html`, `anAnnouncement`, `anAnnouncer`) parámetros.

- **on: anAnnouncement of: anAnnouncer update: anId sending: aSelector** Realiza la misma actualización que utilizando un bloque, pero permite el reuso de mensajes de renderizado. Este mensaje puede tener los mismos argumentos que el bloque de renderizado.

Insertions. A diferencia de los *updaters*, los *insertions* no actualizan información en la página Web sino que agregan contenidos DOM. El objetivo de los *insertions* es el de insertar, en cuatro posiciones diferentes, nuevos elementos a partir de una posición específica de la página. Para hacer esto es necesario suministrarle el elemento DOM al cual se le van a adosar las inserciones.

- **on: anAnnouncement of: anAnnouncer insertIn: anId at: aSymbol callback: aBlock** Inserta el código HTML generado mediante la ejecución del bloque, cualquiera en la posición descrita por `aSymbol`. Este símbolo puede ser `#before`, `#top`, `#bottom` o `#after` y el bloque puede tener los mismos parámetros que en el caso de los *updaters*.

- **on: anAnnouncement of: anAnnouncer insertIn: anId at: aSymbol sending: aSelector** Al igual que el anterior inserta el HTML generado pero en este caso es generado por la ejecución del mensaje `aSelector`.

Scripting. Los dos primeros tipos de mensajes son muy efectivos para diferentes tipos de situaciones donde se desee actualizar o insertar contenido en la página Web cada vez que ocurre un cambio en el modelo, utilizando siempre lenguaje Seaside. El problema aparece cuando queremos modificar contenido de la página Web utilizando exclusivamente Javascript. Los escenarios donde esto podría ocurrir son aquellos que manipulan objetos Javascript que están en el navegador Web, como pueden ser nuevos widgets HTML codificados en Javascript (sliders, drag and drop, tablas, etc.), o inclusive efectos visuales [78], manipulación de atributos HTML, Google Maps, etc.

- **on: anAnnouncement of: anAnnouncer javascriptCallback: aBlock** Transfiere al navegador Web el Javascript generado por el bloque `aBlock`. Este bloque al igual que en el caso del requerimiento puede recibir cero, uno (`anAnnouncement`) o dos (`anAnnouncement`, `anAnnouncer`) argumentos.

- **on: anAnnouncement of: anAnnouncer javascriptSending: aSelector** Al igual que el caso anterior se envía Javascript pero en este caso como resultado de la ejecución del mensaje `aSelector`. Puede recibir los mismos parámetros que el bloque.

Para poder visualizar cómo usar alguno de los mensajes provistos por la capa **Observer**, basémonos en el ejemplo de una de las componentes más tradicionales de Seaside: el `WebCounter` [5]. En nuestra reimplementación del ejemplo tenemos una componente visual `Counter` el cual mirará a un modelo compartido que tendrá un número, llamado `CounterModel`. El `#renderContentOn:` de esta componente será tan sencillo como:

```
Counter>>renderContentOn: html
    html div
            id: 'number';
            with: (self model number)
```

lo que provocará que se dibuje una página Web que mostrará el número del modelo, pero cualquier cambio que se haga en el mismo no será reflejado en el navegador Web. Esto es porque el modelo no conoce a las múltiples vistas ni tiene forma de propagar su cambio a ellas, las cuales están mostrando interés en su aspecto `#number`. La forma para enviar los nuevos valores, es que la vista los empuje a través de la conexión, pero para eso la vista debe saber cuando cambia el modelo.

[5] Ver ejemplo en http://book.seaside.st/book/getting-started/pharo-squeak/first-component

Utilizando un mensaje de esta nueva capa podemos ahora definir una actualización para el tag DIV cuyo ID es *number*, de forma tal que cada vez que se modifique el número del modelo pueda ser refrescada la vista Web que hemos creado. Para definir esta actualización utilizaremos el mensaje `#on:of:update:callback:` en el método `#initialize` de nuestra componente `Counter` de la siguiente manera:

```
Counter>>initialize
    super initialize.
    self model: CounterModel default.
    self
        on: ValueChangedAnnouncement
        of: self model
        update: 'number'
        callback:
            [:html |
            html text: self model number.
            ]
```

y por último en el modelo debemos asegurarnos que se propage el `ValueChangedAnnouncement` cada vez que cambie su valor

```
CounterModel>>increase
  self count: count + 1.
  self announce:
    (ValueChangeAnnouncement with: self count).
```

de esta forma definimos que cada vez que el modelo anuncie `ValueChangeAnnouncement`, la componente `Counter` actualizará el elemento DOM cuyo id es *number* con el bloque de renderizado que se envía como argumento. En la Figura 5.5 se observa el diagrama de secuencia que `CounterModel` ejecuta cada vez que el método `#increase` es llamado.

Gracias a la capa **Observer** definir actualizaciones visuales entre el modelo y los clientes Web es sencillo, utilizando el mismo lenguaje que se usa tanto para crear el modelo como las componentes, sin la necesidad de encargarse del mantenimiento de las dependencias, debido a que este fue uno de los principales objetivos de la capa **Observer**. A continuación, se explicará en profundidad cómo se logró tanto la forma de dibujar como la abstracción del mecanismo de dependencias.

Todo en un mismo idioma

El primer objetivo de esta capa fue abstraer el uso de Javascript a la hora de escribir actualizaciones para las aplicaciones Web. Esto fue conseguido gracias

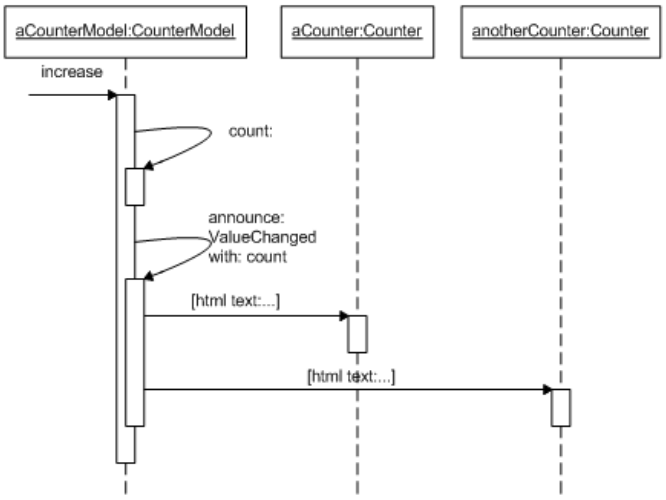

Figura 5.5: Diagrama de secuencia luego que el CounterModel incrementa en 1 su valor

al port realizado para Seaside de la librería Javascript script.aculo.us [79, página 11]. Este framework permite definir actualizaciones (vía Ajax) de distintos tipos utilizando la misma sintaxis que es usada en Seaside, para luego ser codificadas en Javascript, y así actualizar correctamente el navegador Web (para ver mayores detalles ver el Apéndice A). Esta abstración de cómo implementar distintos tipos de llamados Ajax (`Updaters`, `Insertions`, `Requests`, etc.), nos facilitó las bases para que la capa **Observer** pueda brindar un protocolo mucho más abstracto que la capa **PushScript**.

Volviendo al ejemplo del chat explicado en la Sección 5.2.2, cada vez que se quería agregar un nuevo mensaje en la lista de mensajes se debía invocar a una función Javascript que dado dos textos (nombre usuario, mensaje) insertara una nueva línea a la lista, y luego enviar ese llamado desde el servidor al cliente (utilizando el método `#pushScript:`) para que dicha función con los datos a actualizar sea ejecutada. Esto provoca, como se mencionó anteriormente, la mezcla de de código Javascript con Smalltalk para realizar actualizaciones. En cambio, si uno quisiera escribir la misma actualización pero utilizando esta librería sería algo de la siguiente forma

```
MetChat>>update: aSymbol with: aMessage
    self pushScript:
        (SUInsertion bottom
            id: 'messageList';
            with: [:html|
                self renderMessage: aMessage
                on: html
```

```
                ]
            )
        ...
```

donde el mensaje `#renderMessage: aMessage on: html` es el mismo método que se usa para renderizar un mensaje en el `#renderContentOn:`. Como se ve en este ejemplo, el uso de este port permite escribir las mismas actualizaciones que antes pero sin la necesidad de escribir Javascript y con la posibilidad de reusar código, lo que facilita la codificación de este tipo de aplicaciones, como es a nivel de lectura, mantenimiento, capacitación, etc. Si codificáramos el método `#renderMessage: aMessage on: html` quedaría:

```
MetChat>>renderMessage: aMessage on: html
    html div with:[
        html strong: aMessage user , ' dice: '.
        html text: aMessage body.
    ]
```

este fragmento de Seaside imprimirá el mismo HTML que es generado en la capa **PushScript**,

```
<html>
  ...
  <body> ...
    <div id='messageList'>
        ...
        <div><strong>Usuario_1</strong> dice: hello</div>
    </div>
  </body>
<html>
```

pero en vez de codificar HTML via Javascript, aquí utilizamos la sintaxis Seaside. No solo trae ventajas a nivel de lectura (menos y más legible código), sino que además podemos editar de forma ágil, refactorizar, debuggear y corregir errores con las herramientas del sistema Smalltalk donde estemos codificando.

Es por esto, que hablar todo en el mismo idioma optimiza la forma de desarrollar soluciones, no preocupándose del cómo y abocarse directamente a implementarlo.

Manejo de dependencias

El segundo objetivo de esta capa era proveer la posibilidad de crear dependencias entre el modelo y las componentes Meteoroid para definir actualizaciones de forma simple y prolija. Como se explica en el Apéndice B, para crear estas

aplicaciones es necesario el uso del patrón Observer. De las implementaciones provistas por Visualworks, Announcements resulta la mejor debido a que representa los eventos como objetos que pueden tener estado y comportamiento.

Cada vez que se define una actualización utilizando el protocolo provisto por la capa **Observer**, entre un modelo y una vista, las dependencias necesarias para dicha actualización no son creadas en ese mismo momento, de hecho son creadas en respuesta a un requerimiento del cliente. Pero para entender esto, primero debemos explicar los `DeferredAnnouncementsDeclarations`.

Un `DeferredAnnouncementsDeclaration` contiene la información necesaria para crear las dependencias entre un modelo y su vista, además de realizar una acción específica ante un evento en particular. Para hacer esto, los `DeferredAnnouncementsDeclarations` poseen en sus estados, el evento al cual se le va a prestar atención (un `announcement`), el modelo (un `announcer`), el DOM ID a actualizar (un `id`) y un bloque para dibujar el nuevo HTML (un `block`). Hay diferentes tipos de `DeferredAnnouncementsDeclarations`: `DeferredUpdateDeclaration`, `DeferredInsertionDeclaration`, `DeferredJavascriptDeclaration` y `DeferredRequestDeclaration` dependiendo si se desear realizar una actualización, una inserción HTML, una inserción de Javascript o un requerimiento respectivamente. Lo bueno de estos objetos (ver Figura 5.6), es que encapsulan el comportamiento de ligar un modelo con su vista, así como también tienen la lógica para poder eliminar ese vínculo cuando lo requiriese. Gracias a este desacople de comportamiento, es posible conectar y desconectar las dependencias en diferentes momentos críticos de la conexión. Esto último será explicado en breve.

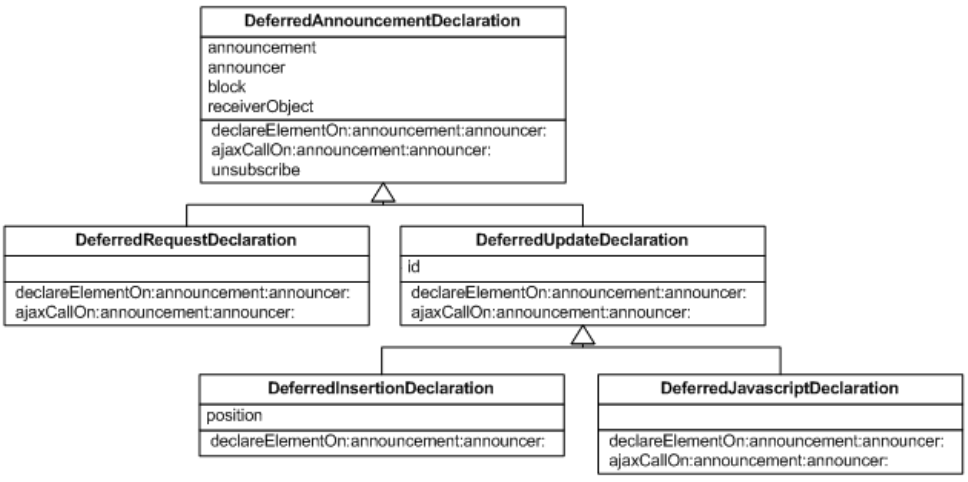

Figura 5.6: Diagrama de la jerarquía DeferredAnnouncementsDeclaration

Como se mencionó, las dependencias entre la vista y el modelo no son creadas por una acción en el servidor al momento de renderizar o crear las componentes,

sino que a partir de un llamado del cliente la vista se agrega como dependiente del modelo. Esto fue desarrollado así, por un problema fundamental de los botones `back` y `forward` de los navegadores Web. Estos botones permiten fácilmente moverse por el historial de navegación del usuario, aprovechando la caché del navegador. Este último detalle es muy importante, ya que cachear una página implica que los eventos de `back` o `forward` del navegador Web no realizarán un requerimiento HTTP para pedir por la página, sino que mostrarán la información obtenida en el primer requerimiento. Dado este contexto, no podíamos entonces conectar el modelo con la vista en el renderizado de las componentes, puesto que si el usuario navega a través de las páginas cacheadas las componentes no serán creadas ni inicializadas en cada `back` o `forward`, y por lo tanto las mismas no serán dependientes del modelo la próxima vez que sean visitadas vía caché.

La decisión entonces de realizar la conexión a través de un llamado Javascript, es porque a pesar de que una página se cachee (mismo contenido HTML), los scripts dentro de la misma serán ejecutados cada vez que se muestre la página esté o no esté cacheada [73], y por lo tanto las conexiones entre el modelo y la vista serán propiamente inicialazas. Ver Figura 5.7, donde se ilustra lo anterior.

Los `DeferredAnnouncementsDeclarations` entonces, son invocados vía Javascript desde el cliente para que creen la conexión y en caso que amerite, romper las viejas conexiones (este caso, se produce si una persona va y vuelve muchas veces sobre la caché, crearía dependencias obsoletas). Estos objetos creados al momento de la definición (una única vez por componente) y son guardados en una colección que toda componente `MeteoroidObserver` posee. Una vez que todo el proceso de renderizado de la componente ha terminado, el navegador Web realiza los llamados Javascript, creándose todas las dependencias definidas por los objetos `DeferredAnnouncementDeclarations`, generando los códigos necesarios para realizar actualizaciones, inserciones, etc. Esto fue posible modificando nuevamente el método `#renderWithContext: aRenderingContext`, agregando luego del llamado al *hook method* `#renderContentOn:` un llamado al método `#createJSObservers:`, el cual fue definido de la siguiente manera:

```
MeteoroidObserver>>createJSObservers: html
  html script:
    (html updater
      callback: [:renderer |
      self createObservers: renderer
      ]
    ) asJavascript
```

Como se observa, este mensaje agrega una línea de código Javascript en la componente Web, la cual hará un llamado Ajax. En el llamado a `#createObservers:` removerá cualquier dependencia que haya quedado colgada (recordar los botones de `back` y `forward`) y se crearán las nuevas dependencias con la información que

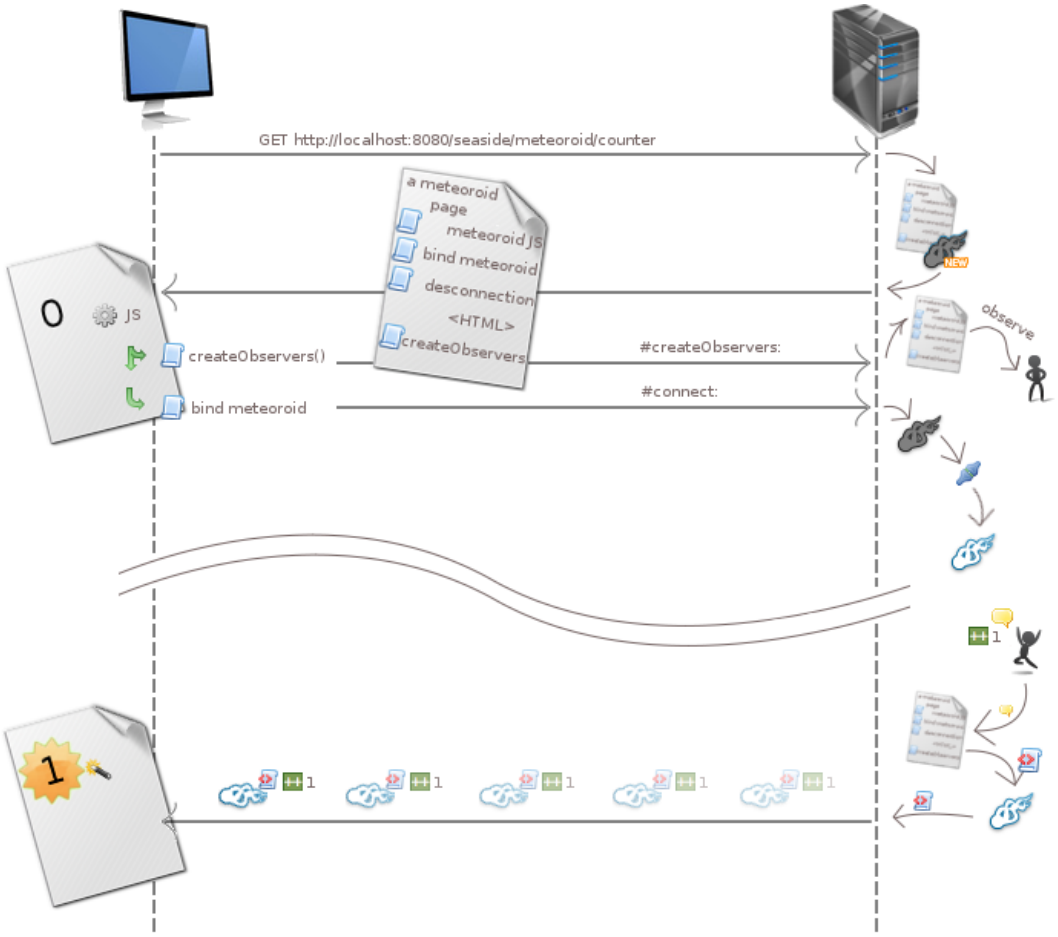

Figura 5.7: Secuencia de conexión de Capa Observer

fue provista en la declaración.

Una vez creadas las dependencias, cada vez que haya alguna notificación del modelo, la componente `MeteoroidObserver` se actualizará automáticamente. Adicionalmente, cada vez que el usuario termine de utilizar la aplicación y decide cerrarla o irse a otro sitio las dependencias son removidas automáticamente por el framework. Como se explicó anteriormente las dependencias se crean cuando el cliente realiza un llamado Javascript, disparado por el evento `document.onLoad()`, al servidor. Para romper las dependencias utilizamos un mecanismo similar, el evento `document.onUnload` del documento HTML. Este evento es activado cuando el navegador Web es cerrado, o el usuario cambia la navegación a otra página. En particular, pusimos un pequeño Javascript que llama al mensaje `#hardDisconnect` de cada componente de la aplicación Web, el cual termina en el mensaje `#undeclareDeferredAnnouncements`. Este mensaje fue implementado de la siguiente forma

```
MeteoroidObserver>>undeclareDeferredAnnouncements
    self deferredAnnouncements
        do: [:deferredAnnouncement |
           deferredAnnouncement unsubscribe]
```

como se observa, todas las dependencias creadas por los `DeferredAnnouncementDeclaration` son removidas en cada componente mediante el mensaje `#unsubscribe`.

Reimplementando el chat, con Observers

Retomando el ejemplo del chat explicado en la Sección 5.2.2, se describirá a continuación los pasos para reimplementarlo utilizando las ventajas de esta capa. En el `#initialize` de la componente `MetChat` tendremos algo de la forma

```
MetChat>>initialize
    super initialize.
    ...
    self
        on: NewMessageAnnouncement
        of: self chatModel
        insertIn: 'messageList'
        at: #bottom
        sending: #renderMessageOn:announcement:
```

donde la componente sabe que debe crear una dependencia con el modelo (el `chatModel`) esperando por alguna notificación de mensaje nuevo (un `NewMessageAnnouncement`) y cuando eso ocurra deberá insertar dicho mensaje

al fondo de la lista (usando el mensaje `#renderMessageOn:announcement:`). El mensaje en el modelo será algo como

```
Room>>sendMessage: aString from: anUser
  "guardar el mensaje.."
  self announce:
    NewMessageAnnouncement new: aString from:
       anUser)
```

donde la notificación llegará a la instancia de `MetChat` correspondiente, la cual insertará al fondo de la lista de mensajes el resultado de ejecutar el mensaje `#renderMessageOn:announcement:`

```
MetChat>>renderMessageOn: html announcement:
   aNewMessageAnnouncement
    (html div)
        class: 'message';
        with:[
            html span
                with: aNewMessageAnnouncement user.
            html text: ' dice: '.
            html span
                with: aNewMessageAnnouncement
                   message].
```

terminando este flujo en el cliente, donde será insertada la nueva línea en el fondo del DIV cuyo ID es *messageList*. A diferencia del ejemplo con el mecanismo de símbolos, ahora la dependencia que existe entre el modelo y la vista es manejada por la capa **Observer** de forma transparente al desarrollador. Enfocándose sólo en qué se quiere renderizar, dónde y cuándo se desea hacerlo.

Resumen

Como se vio a lo largo de esta sección, esta capa provee una forma simple de crear aplicaciones Comet sin la necesidad de enfocarse en los aspectos técnicos del manejo de conexiones ni actualizaciones entre el modelo y la vista, permitiendo que el desarrollador diseñe sus aplicaciones Web de una forma muy similar a las aplicaciones Web que generaba utilizando solamente Seaside.

La capa **Observer** tiene además un protocolo para que la vista pueda ser sincronizada automáticamente con su respectivo modelo, usando las ventajas del patrón arquitectural MVC. Para poder facilitar su uso, también se desacopló el mecanismo de dependencia de forma tal que `Meteoroid` se encargue de acoplar y/o desacoplar componentes con sus modelos, haciendo que el único objetivo sea el de representar la información en el navegador Web, el resto corre por `Meteoroid`.

5.2.4. Web Live Widgets

Como se vio en la capa **Observer**, crear aplicaciones Web con Meteoroid es muy simple y eficaz, pero sería interesante conseguir un mayor nivel de abstracción. La metodología utilizada por VisualWorks para realizar las aplicaciones de escritorio (ver Apéndice B para explicación detallada) es una forma muy interesante de crear aplicaciones. Los GUI widgets se conectan a los aspectos del modelo que les interesan mediante `ValueModels` que proveen de una forma genérica y simple un protocolo de notificaciones (mediante el llamado a `changed:#value`) y acceso (mensajes `#value` y `#value:`). De esta manera se pueden desarrollar widgets considerando que cualquier modelo entenderá el protocolo `#value` y a su vez los modelos también pueden ser desarrollados sin importarle el nombre que deseen ponerle a sus variables.

Esta última capa tiene como objetivo implementar una funcionalidad equivalente a la utilizada por VisualWorks pero adaptado a las aplicaciones Web. Provee un conjunto de widgets HTML que son asociados a aspectos del modelo de aplicación y luego actualizados automáticamente por Meteoroid cuando el modelo se modifica, de la misma forma que sucede con los widgets de las aplicaciones de escritorio.

Funcionalidad provista

La capa **Web Live Widgets** provee un conjunto de widgets HTML (*live widgets*), que a partir de un modelo se encargan de pintarse cada vez que sea nececesario. Para poder lograr esto, esta capa depende fuertemente de dos clases:

- `ValueModelWrapper`. Wrappers [6] [75, Decorator pattern] de los `ValueModel`, mejoran el funcionamiento de dependencias gracias al uso del framework `Announcements` (los `ValueModel` usan sólo el mecanismo de dependencias con símbolos).

- `LiveTag`. Wrappers de los tags estándares de Seaside, permiten que elementos HTML (DIVs, SELECTs, INPUTs, etc.) puedan ser cometificados vía un `ValueModelWrapper`.

La siguiente Figura 5.8 refleja el apilamiento de capas, que permiten tener tags vivos en los navegadores Web.

Para la utilización de estos *live widgets* se agregaron una serie de métodos en la clase `WARenderCanvas`, permitiendo que el desarrollador pueda crearlos de la misma forma que lo hacía cuando creaba un tag HTML común. El protocolo agregado, una vez invocado, crea un tag HTML asociándolo al `ValueModelWrapper` que recibe como argumento. Los mensajes posibles son:

[6] También conocido como Decorators, el patrón Wrapper permite agregar responsabilidad a un objeto en vez de a toda la clase de forma dinámica.

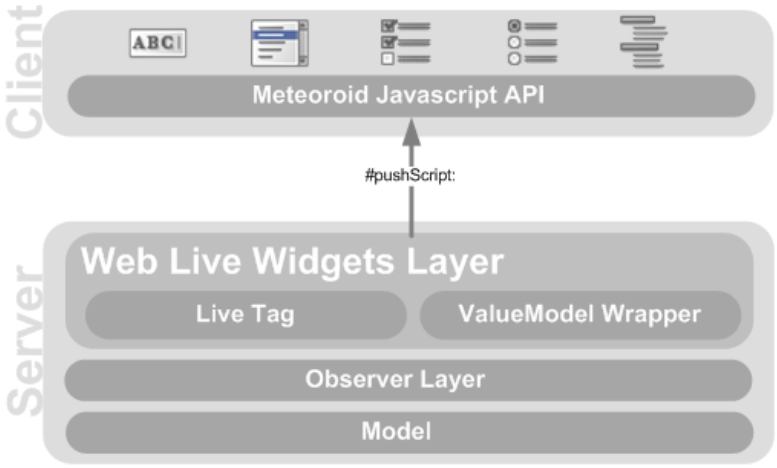

Figura 5.8: Funcionamiento de la capa Web Live Widgets

- **liveDivFor: aValueModelWrapper**
- **liveSpanFor: aValueModelWrapper**
- **liveInputFor: aValueModelWrapper**
- **liveTextAreaFor: aValueModelWrapper**
- **liveOrderedListFor: aSelectionInListWrapper**
- **liveUnorderedListFor: aSelectionInListWrapper**
- **liveSelectFor: aSelectionInListWrapper**
- **liveCheckboxCollectionFor: aMultiSelectionInListWrapper**
- **liveRadiobuttonCollectionFor: aSelectionInListWrapper**
- **liveDivListFor: aSelectionInListWrapper**

Si retomamos el ejemplo del `Counter` resulta aún más sencillo implementarlo utilizando esta capa. Por un lado se debe crear el `ValueModelWrapper` necesario dentro del `#initialize`, para interesarse en el contador compartido

```
Counter>>initialize
    super initialize.
    self model: CounterModel default.
    self valueModel:
            (ValueModelWrapper
                with: self model
```

```
                    aspect: #number).
```

esto permite que cada vez que el modelo cambie, la variable de instancia *valueModel* del `Counter` (un `ValueModelWrapper`) sea notificada. Por el otro lado se debe renderizar el *DIV vivo* utilizando el protocolo provisto por esta capa en el método `#renderContentOn:`

```
Counter>>renderContentOn: html
    ...
    html liveDivFor: self valueModel
```

de esta forma, cada vez que el modelo realice un `changed:#number` se actualizará automáticamente el DIV localizado en el navegador Web vía Comet. Esta nueva funcionalidad permite crear el mismo ejemplo pero mucho más sencillo, sin preocuparse cómo debe renderizarse los elementos Web, puesto que esa es la tarea de la capa **Web Live Widgets**, promover el uso de widgets de alto nivel.

Veamos ahora, cómo se podría tener un *SELECT vivo* con una lista que mute. Supongamos que tenemos la clase `ShopCart`, la cual esta compuesta por una lista de ítems (para simplificar el ejemplo, supongamos que cada ítem es un string). En un comienzo, esa lista estará vacía y se inicializará en el método `#initialize` del objeto `ShopCart`

```
ShopCart>>initialize
  ...
  self items: List new.
```

en particular, un objeto `List` anuncia a sus dependientes de forma automática frente diversas situaciones: adición, edición, remoción, etc. Con eso bastará para agregar y remover elementos a la lista de ítems de un `ShopCart`. Además, un objeto `ShopCart` entenderá mensajes para agregar y remover ítems

```
ShopCart>>addItem: aString
  self items add: aString.
```

el cual agrega un elemento a la lista, la cual propagará una notificación a sus dependientes (remover es de similar características). En la vista tendremos una componente, `ShopCartView` el cual debe definir dos métodos, el `#initialize`

```
ShopCartView>>initialize
  self model: ShopCart new.
  self selectionInList:
      (SelectionInListWrapper
        with: self model items)
```

el cual creará el `SelectionInListWrapper` asociado a la lista. Luego, en el `#renderContentOn:` se debe dibujar el *LiveSelect*

```
ShopCartView >>renderContentOn: html
  ...
  html liveSelectFor: self selectionInList.
```

Con esto implementado, cada vez que se llame al método `#addItem:`/`#removeItem:` de un `ShopCart`, el `LiveSelect` será actualizado instantáneamente. Notar que no existe ningún tipo de definición sobre qué aspecto nos interesa del modelo, `MeteoroidObserver` se encarga de manipularlas por completo.

Para desarrollar los *live widgets* sobre las capas anteriores de **Meteoroid** se implementaron dos jerarquías de objetos, por un lado los `ValueModelWrappers` que añaden comportamiento a los actuales `ValueModel`, y por el otro los `LiveTags` que permiten la creación de widgets actualizables. Ambas jerarquías serán detalladas a continuación.

ValueModelWrapper

Se desarrolló un conjunto de clases que heredan de `ValueModelWrapper` para agregar funcionalidad a la jerarquía de `ValueModel` existente. Estos nuevos wrappers permiten propagar notificaciones usando el framework Announcements, ya que los `ValueModels` sólo soportan la implementación del patrón Observer que utiliza símbolos. La forma en que trabajan los wrappers, es capturando las actualizaciones de los `ValueModels` que enmascaran, para lanzar `Announcements`. Existen varias clases que wrappean diferentes `ValueModels`:

- **ValueModelWrapper.** Esta clase fue diseñada para wrappear a `ValueHolder` y a `AspectAdaptor`. Cada vez que alguno de estos objetos notifican un cambio mediante el símbolo `#value`, la instancia de `ValueModelWrapper` que esta enmascarándolo anuncia un cambio disparando un `ValueChangedAnnouncement`.

- **SelectionInListWrapper.** Este wrapper enmascara al `SelecionInList` ofreciendo las notificaciones genéricas para modificaciones en listas utilizando el framework de Announcements. De esta forma se notifica el cambio de la lista y el cambio de selección, agregado, modificación y el borrado de un elemento.

- **MultiSelectionInListWrapper.** Esta subclase de `SelectionInListWrapper` agrega notificaciones de Announcements al objeto `MultiSelectionInList` permitiendo mantener el control de múltiples selecciones sobre una lista.

- **WebValueModelAnnouncement.** Este objeto tiene un comportamiento equivalente al de un `AspectAdaptor` pero en vez de estar interesado en notificaciones de la implementación vieja (mediante un símbolo), espera un `Announcement`.

En la Figura 5.9 se observa cómo funciona cada wrapper

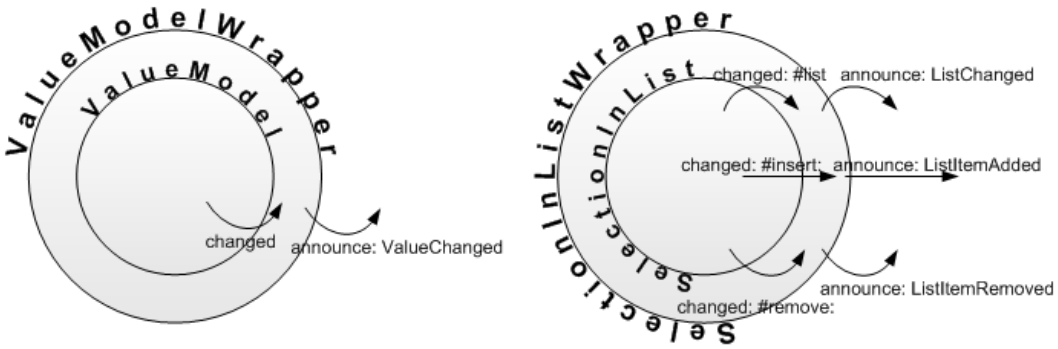

Figura 5.9: Ejemplo de dos tipos de Web Live Widgets

Para poder entender el uso de los **Web Live Widgets**, pensemos en una vista cuyo dominio es una persona. Tendríamos la clase `Person` que modeliza a una persona, con algunas variables de instancia, como por ejemplo *name*. En la vista, `PersonView` podemos crear un `ValueModelWrapper` que este interesado en el nombre de la persona y luego esperar las notificaciones desde el `ValueModelWrapper` para actualizar la misma, cada vez que el modelo `Person` notifique que ha cambiado en su nombre. Al igual que en la capa **Observer** donde dentro del `#initialize` se definían las dependencias y bloques de pintado, ahora la componente debe crear los `ValueModelWrapper`, como se observa

```
PersonView>>initialize
    ...
    self aspectAdaptorWrapper:
            (ValueModelWrapper
                with: self user
                aspect: #name).
    self aspectAdaptorWrapper when:
       ValueChangedAnnouncement send: #
       userNameChanged to: self
```

estas líneas crearán un `ValueModelWrapper`, que wrappeará un `AspectAdaptor` interesado en el nombre de la persona. La vista, a su vez, se hará dependiente del anuncio genérico que notifique el `ValueModelWrapper`. Si ahora nos enfocamos

en el modelo, cada vez que se modifique el nombre del objeto `Person`, propagará el cambio de la siguiente manera

```
Person>>name: aString
    name := aString.
    self changed: #name
```

este `changed: #name` propagará una notificación a sus dependientes, entre ellos el `AspectAdaptor` el cual lanzará un `changed: #value`. Con este último `changed:`, el `ValueModelWrapper` anunciará dicho cambio con un `ValueChangedAnnouncement`, finalizando en el mensaje `#userNameChanged` en la instancia `PersonView`. Como se explicó, con esta nueva forma se pueden crear notificaciones utilizando ambos frameworks de notificación de una manera sencilla, el modelo cambia y notifica para que Meteoroid se encargue de propagar el nuevo valor al navegador Web. Si se observa la siguiente Figura 5.10 se entenderá el mecanismo de forma gráfica de cómo un `ValueModelWrapper` puede ser utilizado para unir un modelo, junto con un `ValueModel` y un mensaje de actualización que existe en la vista `PersonView`.

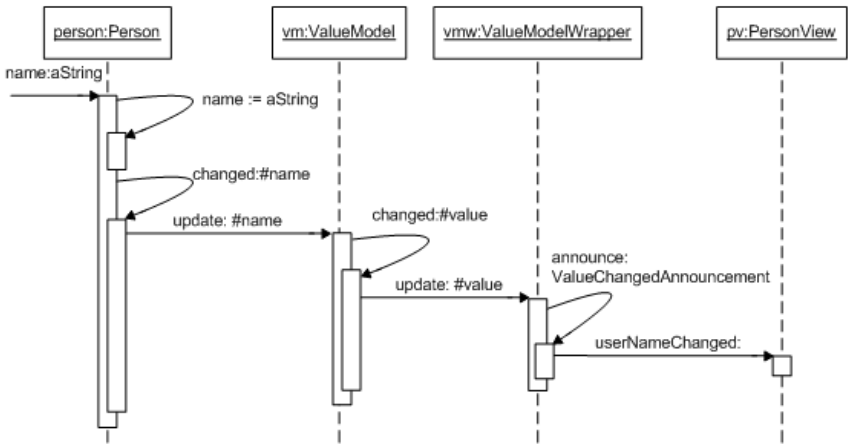

Figura 5.10: Diagrama de secuencia Web Live Widgets

LiveTag

Seaside provee un extenso protocolo para poder utilizar todos los tags HTML, pero esto no era suficiente para **Meteoroid** y por tal motivo se diseñaron los **Web Live Widgets**. Para poder extender los widgets normales y dar origen a los *live widgets*, parte central de esta nueva capa, fue necesario crear los `LiveTag`, widgets HTML actualizables de forma automática. Estos fueron diseñados en una jerarquía cuya superclase es la clase abstracta `LiveTag`, los cuales son wrappers

de las clases encargadas de representar los tags HTML en Seaside pero con dos nuevos detalles:

- Tienen la capacidad de conectarse con un `ValueModelWrapper`.
- Cada vez que el `ValueModelWrapper` se modifique ante un evento, el `LiveTag` entiende ese tipo de evento y dispara las actualizaciones necesarias para que el navegador Web refleje de forma precisa los cambios del modelo.

Internamente, los `LiveTag` están compuestos por varios objetos que hacen posible su funcionamiento:

page Contiene una componente subclase de `MeteoroidObserver` donde el widget HTML se dibujará y donde se realizarán las dependencias necesarias para que se actualice automáticamente.

renderer Objeto por el cual podemos renderizar HTML en las páginas.

tag Es un objeto `WATag` encargado de dibujar un tag HTML estándar en Seaside. En este caso tendrá la subclase que corresponda al tag que esté wrappeando.

valueModel Este es un `ValueModel` del cual se extraerá la información necesaria para actualizar la vista, si hubo algún cambio en el modelo, o el modelo si hubo algún cambio en la vista.

De forma similar a los widgets de las aplicaciones de escritorio quienes delegan responsabilidad en los `ValueModel`, los `LiveTags` fueron desarrollados delegando el mantenimiento de la información a manos de los `ValueModelWrappers` y de esta manera centrarse en cómo mostrar la información de forma adecuada en las componentes Web. Para lograrlo, los `LiveTag` dependen de un `ValueModelWrapper` al cual acceden mediante el mismo protocolo genérico que proveía `ValueModel`.

Para poder implementar estos *live widgets* fue necesario extender y modificar algunos aspectos. En particular, nos referimos a la forma en la cual los RADIOBUTTONs (`WARadioButtonTag`) y CHECKBOXes (`WACheckboxTag`) son usados en Seaside, ya que estos tags son vistos y utilizados como únicos cuando en realidad suelen formar parte de un conjunto. Por ejemplo, si queremos mostrar un conjunto de RADIOBUTTONs a partir de una colección de usuarios en Seaside, deberíamos realizar una componente la cual defina un grupo (`group`), para luego iterar sobre una lista (supongamos una lista de usuarios) y en cada pasada renderizar el RADIOBUTTON

```
ExampleComponent>>renderContentOn: html
    | group |
    ...
    group := html radioGroup.
    self users
```

```
            do:[:user |
                group radioButton
                    callback: [ self selectedUser:
                        user].
                html text: user name.
            ]
    ...
```

pero en estos casos sería mejor si el tag RADIOBUTTON fuera visto como una colección en sí misma. Por este motivo hemos creado tags que toman tanto a los RADIOBUTTONs (en el caso de `RadioButtonCollectionTag`) y a los CHECKBOXs (en el caso de `CheckBoxCollectionTag`) como colecciones, donde a partir de una lista se generan los diferentes elementos. Si reimplementamos el ejemplo anterior utilizando los nuevos tags, podríamos hacer lo siguiente

```
ExampleComponent>>renderContentOn: html
    ...
    html radioButtonCollection
        list: self users;
        labels:[:user | user name];
        callback:[:user | self selectedUser: user]
    ...
```

al tratarlo como una colección, no solo hace el código más simple sino que además nos permite wrappear estos nuevos tags HTML con *live widgets*.

Una vez creada la suite de *live widgets*, se agregó un conjunto de mensajes para accederlas a la clase `WARenderCanvas`. Esta clase es en efecto, el parámetro *html* utilizado dentro del método `#renderContentOn: html`, el cual funciona como un tipo *factory*. Con este nuevo protocolo, ahora es posible acceder a los métodos que fueron descritos en la Subsección *Funcionalidad provista*, los cuales permiten escribir *live widgets* en una forma sencilla, como se observa

```
PersonView>>renderContentOn: html
    ...
    html liveDivFor: self aspectAdaptorWrapper
```

en caso de querer un tag *DIV vivo* cuyo `ValueModelWrapper` sea un `AspectAdaptorWrapper`.

Cada `LiveTag` delega el proceso de renderizado a su tag asociado, y una vez que el tag estándar es dibujado, el `LiveTag` se encarga de crear las dependencias entre su `ValueModelWrapper` y la componente `MeteoroidObserver`, utilizando el protocolo provisto por la capa **Observer**. Cada vez que el modelo modifique el aspecto que le interesa al `ValueModelWrapper`, el tag será actualizado de forma

adecuada, debido a que cada *live widget* sabe como repintarse. En la siguiente Figura 5.11 podemos ver la jerarquía, en la cual cada una de ellas (salvo la clases abstractas *LiveTag* y *LiveCollection*) mapea con un tag HTML de Seaside, donde un *LiveSelect* tiene un *WASelectTag*, un *LiveDiv* tiene un *WADivTag*, etc.

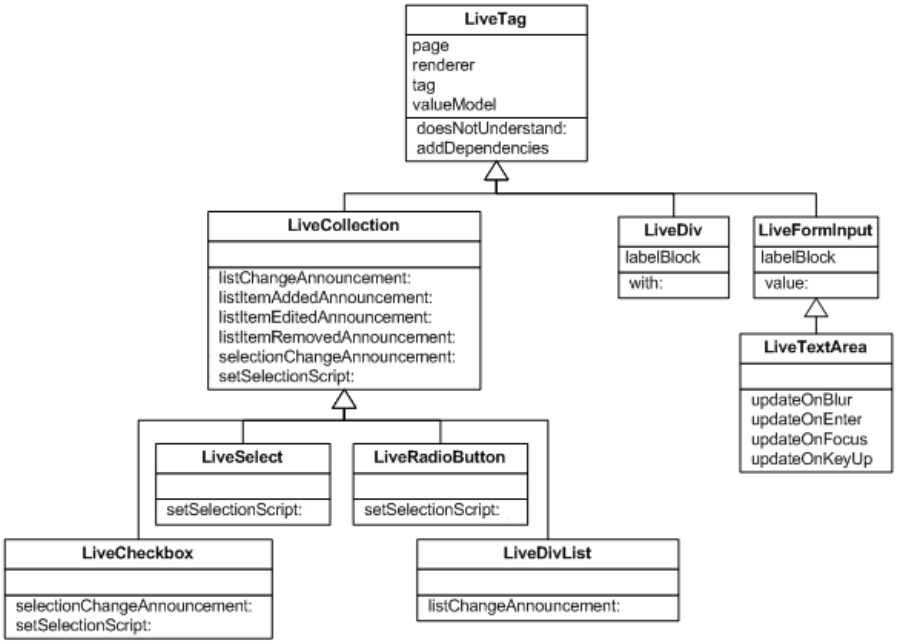

Figura 5.11: Diagrama de la jerarquía LiveTag

Lo interesante de los *LiveTag* es que son transparentes a la hora de interactuar con el proceso de dibujado (recordar que el que realmente escribe HTML es el tag asociado al *live widget*), esto permite agregar propiedades de estilo, efectos, etc. al verdadero tag que renderiza. Esto fue logrado reimplementando el mensaje *doesNotUnderstood:* para que los mensajes de dibujado sean propagados al tag de Seaside cuando corresponda.

Para entender la representación interna de un *live widget* sencillo, veamos como es el `LiveDiv`, un tag DIV normal pero con capacidades Comet. Todos los `LiveTag` definen, en algún punto, las dependencias y en el caso particular del `LiveDiv` esto ocurre en el método `#addDependencies`

```
LiveDiv>>addDependencies
    self page
        on: ValueChangedAnnouncement
        of: self valueModel
        update: self tag id
        callback:
            [:html :ann :announcer |
```

```
                    html text:
                            (self labelBlock value:
                                announcer)
            ]
```

este método describe que, cada vez que el *valueModel*(un`ValueModelWrapper`) notifique el `ValueChangedAnnouncement` (un `Announcement`), se actualice un elemento DOM cuyo ID es *self tag id* de la componente *page* en el navegador Web, utilizando el bloque de renderizado que está en el `#callback:`.

El resto de los *live widgets* varían en el tipo de actualizaciones/dependencias que definen, pero todos mantienen el mismo nivel de simplicidad. Algo en lo que pusimos foco cuando realizamos la capa **Web Live Widget**, fue la abstracción y generalidad que provee el uso de los `ValueModelWrapper`. Notar que no importa que tipo de `Announcement` notifica el modelo, ni los nombres de la variables que posee, todo es manejado utilizando los `Announcements` y los métodos genéricos que provee la capa **Web Live Widget**, ya que cada `LiveTag` sabe cómo debe repintarse en el navegador Web.

Reimplementando el chat, con Web Live Widgets

Para ejemplificar la facilidad de uso de los *live widgets* desde el punto de vista del desarrollador de aplicaciones Web, veamos como reimplementar el ejemplo del chat que se viene desarrollando en capítulos anteriores. Para crear la vista del chat debemos tener dos widgets actualizables:

- Una lista de DIVS encargada de mostrar los mensajes, la cual se actualizará a partir de un `SelectionInListWrapper` interesado por la lista de mensajes del `Room`
- Un SELECT responsable de mostrar la lista de usuarios logueados, que se actualizará dependiendo de la lista de usuarios que posea `Room`, con otro `SelectionInListWrapper`.

El primer paso para obtener nuestro nuevo chat, es crear los dos `SelectionInListWrapper` necesarios para ambas listas en el método `#initialize`

```
MetChat>>initialize
    super initialize.
    self messages: (WebSelectionInList
            with: self chatModel messages).
    self usersList:(WebSelectionInList
            with: self chatModel users)
```

donde el `chatModel`(el modelo) devuelve un `List` [7] en el método `#messages`, el cual posee la lista de mensajes que han sido enviados al chat. Lo mismo ocurre con `#users`, pero aquí la lista posee una colección de usuarios.

El segundo paso a seguir es escribir el renderizado del chat en el método `#renderContentOn:`. El fragmento de código a continuación está concentrado en un solo mensaje, centrado en los **Web Live Widgets**, dejando de lado la parte de maquetación (CSS) para hacer más sencilla la explicación

```
MetChat>>renderContentOn: html
    ...
    (html
        liveDivListFor: self messages)
        labels:
        [:aMessage :renderer |
            renderer strong: aMessage username , '
               dice: '.
            renderer text: aMessage message.
        ].
    ...
    (html
        liveSelectFor: self usersList)
        labels: [:user | user username]
    ...
```

Estos son los únicos dos pasos a seguir para poder crear una aplicación Web que muestre un chat, la cual se actualizará automáticamente sin la necesidad de escribir más código. Cabe destacar que al utilizar la capa **Observer** para crear las dependencias necesarias de los *live widgets*, todas estas son manejadas por el framework, liberando al desarrollador del mantenimiento de las mismas.

Extensiones

Un punto interesante de **Web Live Widgets** es la facilidad para poder extender comportamiento con nuevos widgets. Al estar implementado siguiendo el patrón MVC, el nuevo widget sólo debe centrarse en cómo renderizar las actualizaciones que lleguen.

Un ejemplo de esta facilidad es la lista de DIVs utilizada en el ejemplo del chat. Para este ejemplo queríamos mostrar la lista de mensajes como una colección de DIVs, cada uno formateado de una forma en particular (el usuario en negrita y el mensaje en texto normal). Por este motivo se decidió extender la capa por un widget nuevo que permitiera crear este tipo de comportamiento.

[7] Los `List` son colecciones que disparan cambios en agregar, remover, cambios de índices, etc., útiles cuando se quieren colecciones con notificaciones ante determinados eventos predefinidos.

Lo primero que se debía crear era un tag de Seaside (`DivListTag`), subclase de `WAListTag`, que dibujara una lista de DIVs (tres mensajes para lograrlo). Por el otro lado, a partir de una lista que se debía crear un wrapper, al cual llamamos `LiveDivList`, que asociara un `SelectionInListWrapper` con el nuevo tag. Como este wrapper enmascarara un tag que muestra listas, se creó como subclase de `LiveCollection`, con sólo dos mensajes de renderizado reimplementados funcionaba correctamente.

Otro ejemplo de la facilidad de extensibilidad fue la implementación de una tabla (ver Figura 5.12) que permitiera actualizar la edición de celdas, el agregado y la remoción de filas. Para este caso se utilizó la tabla Javascript DHTMLX Grid [80], por lo que primero se debió crear un tag que permitiera manejar la tabla Javascript siguiendo la forma de Seaside. Este nuevo tag `TableTag` fue creado con 26 mensajes, de los cuales la mayoría son mensajes para setear atributos en la tabla. Luego se creó un `SelectionInTableWrapper`, que lleva control del agregado y borrado de filas en la tabla, así como también controlar la selección y la edición de una celda. Esta clase tuvo origen debido a que en el framework original no existía un `ValueModelWrapper` para tablas, y pudo ser creado con doce mensajes y cuatro `Announcements` que representan los eventos genéricos (agregar, borrar, editar, etc.). Por último, para enlazar este nuevo `TableTag` con un `ValueModelWrapper`, se creó `LiveTable`, que con sólo 5 mensajes permitió hacer que este tag pase a recibir actualizaciones automáticas desde el servidor para todos los eventos que dispara el `SelectionInTableWrapper` asociado.

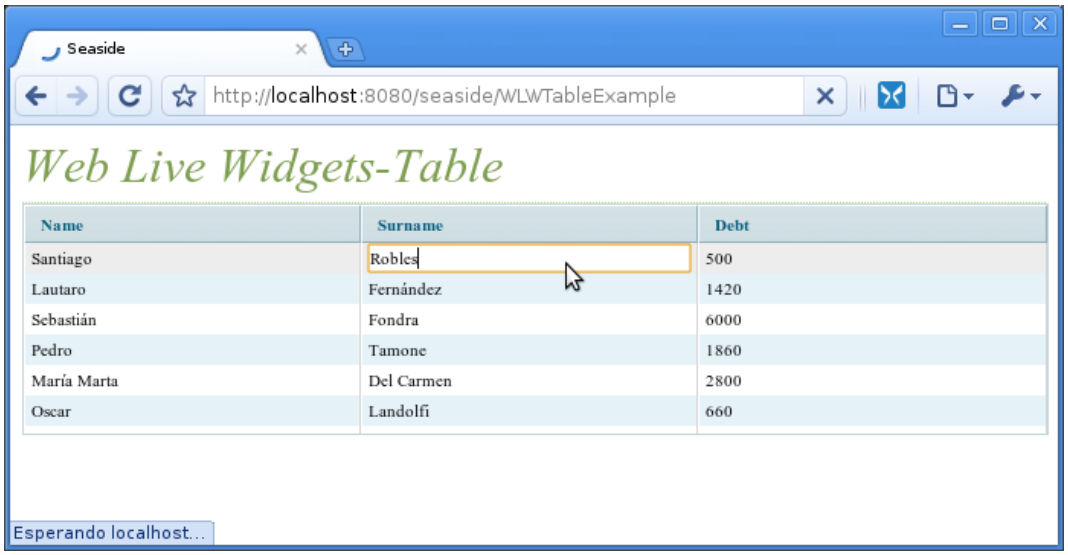

Figura 5.12: Tabla extendida de Web Live Widgets

Como se vio, tanto en el caso de la `LiveDivList`, el cual era una extensión de otros `LiveTags`, como `LiveTable`, que era un nuevo tag desde cero, sólo hace falta muy poco desarrollo para poder crear extensiones, lo que hace que crearlas

sea muy rápido y a su vez muy sencillo.

Resumen

Gracias a la composición de las capas **PushScript** y **Observer** Meteoroid ha alcanzado el mismo nivel de abstracción que posee VisualWorks para el desarrollo de sus aplicaciones de escritorio, permitiendo la creación veloz de aplicaciones Web que poseen un alto grado de interacción entre un modelo y sus vistas (componentes).

Lo interesante de esta capa es la simplicidad, dado que para que una componente `MeteoroidObserver` pueda utilizar los *live widgets*, solo debe tener dos cosas en mente, reimplementar el `#initialize` para poder definir todos los `ValueModelWrapper` que hagan falta, y luego en el `#renderContentOn:` insertar cada *live widget* asociándolo con el `ValueModelWrapper` que corresponda. Otro aspecto interesante de esta capa es su facilidad de extensión, ya que como se describió mediante los ejemplos de `LiveDivList` y `LiveTable`, es muy sencillo crear nuevos widgets, así como también extender el comportamiento de otros.

El objetivo principal detrás de **Web Live Widgets** es facilitar el desarrollo de aplicaciones de escritorio en un contexto Web, a partir de una suite de widgets comunes. Estos widgets además cumplen la particularidad de estar vivos, saber cuándo conectarse con su modelo, cuando romper el vínculo y cómo repintarse. Esto fue conseguido gracias al encapsulamiento en 2 diferentes áreas: los `LiveTag` (tag de Seaside que sabe pintarse) y los `ValueModelWrapper` (wrapper de un `ValueModel` que sabe cuando actualizar a la vista).

5.3. Aplicaciones

Siendo **Meteoroid** un framework nuevo se realizaron diferentes tipos de pruebas para observar los resultados, encontrar errores y tratar de generar tanto feedback como fuera posible para mejorar el mismo. Esta sección contará experiencias que tuvimos a la hora de probar diferentes tipos de contextos (dispositivos móviles, navegadores Web, servicios de hosting, etc.)

5.3.1. Aplicaciones en la Web

A pesar que tener una *conexión Comet* no es algo básico en el gran repertorio de sitios Web que existen, es cierto que si estuviera al alcance de todos, muchos sitios le darían un uso interesante. El objetivo de la tecnología Comet es poder acercar al usuario Web a una experiencia Rich Internet Application (RIA), donde Comet se encarga de habilitar un canal de control entre el servidor y sus clientes Web. Ejemplos claros de estos son las herramientas colaborativas, donde múltiples usuarios pueden acceder a la información para leerla/editarla:

- Wave [81], un cliente rico de correo, con control de cambios, imágenes, audio y video embebido.
- Google Docs [82], suite ofimática para editar documentos en tiempo real.
- Juegos, de cualquiér índole, por turnos, Role Playing Game (RPG), etc.

en síntesis, cualquier aplicación que querramos que se comporte de la misma manera que una de escritorio, debemos tener forma de manipular el navegador Web desde el servidor para notificar los cambios que en él ocurriesen.

Lo bueno de Comet, es que un futuro será una opción nativa dentro del lenguaje de marcado HTML. Actualmente, HTML se encuentra en su versión 4.0, y en su versión 5.0 de draft la cual define un protocolo nuevo: WebSockets. Esto fue detallado en la Sección 3, donde se indicaba la nueva estructura de esta técnica, porqué surgió y las ventajas que trae. Usar Comet hoy en día, es acercarse al nuevo estándar HTML 5.

En el caso particular de **Meteoroid**, lo pusimos a prueba con dos aplicaciones, un *Manejador de Repositorios* y una *Subasta*.

Manejador de Repositorios

Esta herramienta nos permite poder acceder a una imagen remota de VisualWorks para poder administrar los paquetes que se encuentran en el sistema, agregar, remover y editar conexiones a diversos repositorios, y organizar la carga de nuevos contenidos en el sistema. El propósito de este software fue desarrollar una herramienta que permita actualizar versiones en imágenes de Smalltalk que estén en un lugar remoto, de forma gráfica y a la vez amigable al usuario (ver Figura 5.13). Entre las características Comet a destacar, posee un bloqueo exclusivo de forma tal que exista un único administrador (el resto de las personas verán una pantalla de "Trabajando"), un `Transcript`[8] que se ve a lo largo de toda la aplicación y notifica de cambios esenciales en la imagen, junto con visores de versiones, grafos de prerrequisitos, etc.

AuctionHouse

Este proyecto surgió de la necesidad de crear un modelo robusto con una aplicación Web *cometificada*, donde múltiples usuarios pudieran pujar por diferentes objetos dentro de una casa de subastas. Independientemente que el *Manejador de Repositorios* es un buen ejemplo de las capacidades Comet, existe el problema que el dominio de la aplicación es muy focalizado a Smalltalk (y en particular a VisualWorks). Por tal motivo se decidió crear una casa de subastas en línea, donde varios usuarios pudieran interactuar en el mismo espacio y tiempo.

[8]En Smalltalk el `Transcript` es una herramienta de impresión a una consola, similar al `System.out.println();` de Java.

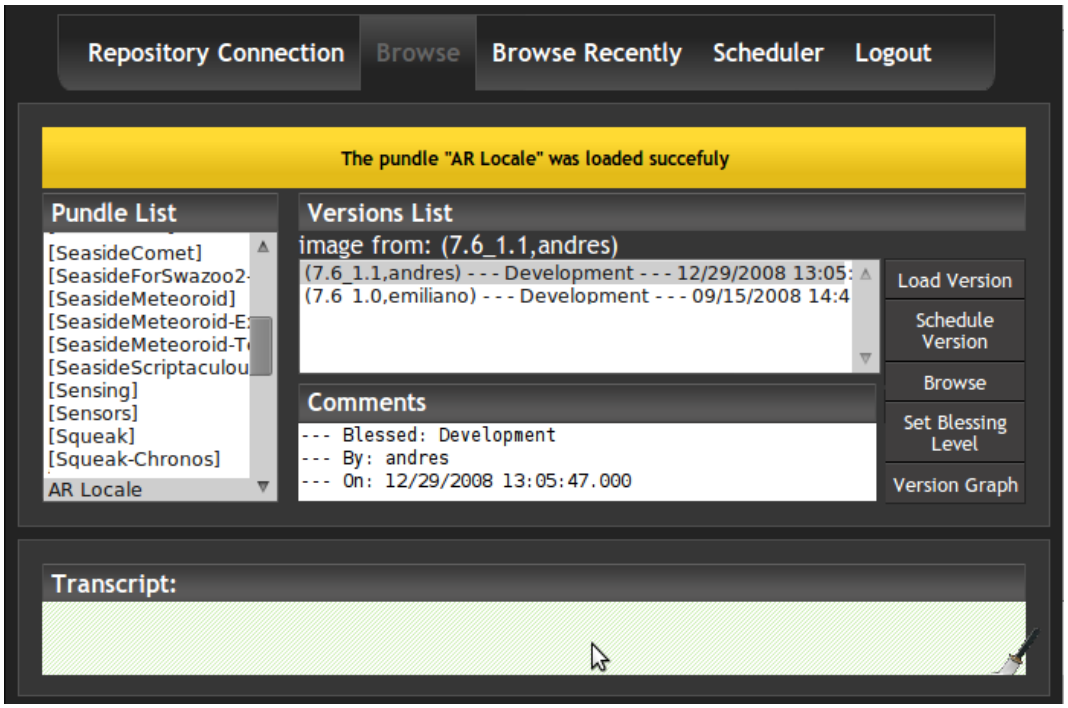

Figura 5.13: Vista luego de cargar un paquete del Manejador de Repositorios

Estos pueden unirse a diversas subastas, pujar por esos ítems, ver cómo aumenta el precio y ver quien es el usuario que va ganando. Las subastas son creadas por un administrador, el cual debe subir una foto del objeto en cuestión más un precio inicial, un nombre, el detalle, y definir cuándo finaliza la subasta. En la Figura 5.14 observamos diferentes componentes visuales de la subasta, en particular focalizando la selección en la cámara "WebCam 8MPX":

- Arriba se encuentra un gráfico de líneas, el cual muestra cómo ha ido aumentando el precio de puja en el tiempo de vida de las subastas. En este caso particular muestra dos líneas porque estamos observando dos subastas al mismo tiempo.
- Abajo se ven dos tabs (uno correspondiente a cada subasta).
- Una componente la cual muestra el nombre del ítem, descripción y una foto.
- Otra componente donde se muestra el precio de puja actual, un cuadro para pujar, y las últimas personas que pujaron por él.

Actualmente existen diversas subastas en línea *cometificadas*, entre las más destacadas podemos encontrar: Swoopo [83], StuffBuff [84], DeACentavos [85], etc. Todas funcionan con un mecanismo similar, un usuario compra una cantidad de pujas y luego las utiliza en los productos que le interesa, en caso de ganar, paga el precio acordado y obtiene el ítem.

Figura 5.14: Vista de una subasta en curso

5.3.2. Uso en Dispositivos móviles

Meteoroid no solo sirve para crear aplicaciones Web que requieran de una actualización en tiempo real, sino que también existen otro tipo de contextos en las que **Meteoroid** puede ser utilizado, como las aplicaciones para dispositivos móviles. El problema es que VisualWorks (ambiente donde nuestro framework está desarrollado) posee un pobre desarrollo de su plataforma para dispositivos móviles, y algunas de las razones que usamos para afirmar lo anterior son:

- Sólo existe una máquina virtual para los sistemas operativos Windows CE [86] y Windows Mobile [87], basados en arquitectura ARM exclusivamente. Esto imposibilita que se puedan desarrollar aplicaciones para nuevos dispositivos móviles que utilizan otros sistemas operativos, como son Android [88], Symbian OS [89], iPhone OS [90], etc.

- No posee una versión minimal de sus imágenes de desarrollo para su utilización, y debe ser el propio desarrollador el encargado de reducir el tamaño ("strip") mediante la remoción de componentes que no sirvan. El problema de stripear una imagen es que dificilmente sea óptima, debido a que es necesario conocer demasiado la ingeniería interna del sistema para poder remover absolutamente todo lo innecesario.

- Posee una GUI desactualizada, sólo se puede crear interfaces con estilo de Windows CE (ver Figura 5.15) provocando que las aplicaciones que corren en Windows Mobile se vean y se sientan de forma muy distinta al resto de las aplicaciones.

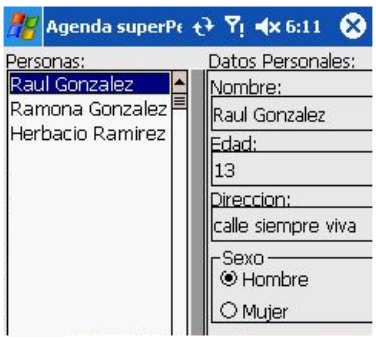

Figura 5.15: Ejemplo de aplicación realizada con VisualWorks para un dispositivo móvil

Para resolver este último punto, es posible la utilización de un servidor Web de forma local junto con **Meteoroid** corriendo en una imagen de VisualWorks en donde se remueva todo el soporte gráfico, usando el navegador Web como la nueva GUI. De esta manera es posible aprovechar el soporte nativo del dispositivo

móvil (el navegador Web), para el desarrollo de aplicaciones mucho más ricas a nivel visual que las que se pueden crear actualmente con VisualWorks, creando aplicaciones actuales, siguiendo el patrón MVC pero permitiendo las ventajas del mundo Web en dispositivos móviles. Además se tiene la ventaja que al estar corriendo un servidor Web dentro del móvil, se pueden acceder a todos los recursos del mismo, y gracias a esto podemos obtener información como las memorias flash, GPS, cámara, discos, etc. (esto está estrechamente ligado a la API provista por el sistema operativo donde se encuentre).

Una aplicación de la propuesta comentada se puede observar en el paper [91]. En este paper se comenta el desarrollo de una arquitectura en donde se combina un servidor Web local (Swazoo y Seaside), **Meteoroid** y una arquitectura Context-Aware [92] para permitir el desarrollo de aplicaciones móviles Web dependientes del contexto (Context-Aware). De la parte Web, se puede aprovechar el diseño de GUIs junto con la posibilidad de utilizar herramientas interesantes (como Google Maps), sumado con **Meteoroid**, el cual agrega la interacción servidor-cliente que existe en las aplicaciones comunes.

Para demostrar su uso se realizó una plataforma de aplicaciones móviles sensibles al contexto. Existe un manejador de aplicaciones móviles (ver Figura 5.16 izquierda), el cual lista las aplicaciones que son disponibles para el usuario. Cada aplicación trabaja como un servicio que puede agregarse o extraerse de la lista de aplicaciones que el manejador posee. Sobre la parte superior existe un área de notificaciones que es utilizado por todas las aplicaciones móviles que se encuentran en ejecución pero pueden no estar siendo visualizadas en un momento dado. Como se ve a la derecha en la Figura 5.16, se creó una aplicación prototipo que a partir del posicionamiento actual[9] muestre dicha posición en un mapa. Esto es fácil de crear ya que es posible codificar aplicaciones que aprovechen componentes Web ya creadas, como Google Maps, con la implementación que el desarrollador de la aplicación requiera.

Como se vio, este puede ser otro tipo de enfoque muy interesante para el desarrollo de aplicaciones en dispositivos móviles debido a su portabilidad y los beneficios que fueron comentados.

5.4. Comparación de frameworks

En el transcurso de este capítulo se describió el framework **Meteoroid**, mostrando los beneficios de su uso. A lo largo de esta sección se realizará una comparación con respecto a otros frameworks, mediante el desarrollo del ejemplo del chat. Uno de ellos será **IceFaces**, debido a que cuando se lo describió en el Capítulo 4 se comentó su beneficio en cuanto a su nivel de abstracción para las actualizaciones. El otro framework elegido será **APE**, que también fue destacado por su capacidad para soportar clientes.

[9] Obtenido del GPS incorporado en el dispositivo.

Figura 5.16: Capturas de aplicación móvil realizada utilizando Meteoroid

Repasando rápidamente, el modelo del chat constará de una clase `ChatRoom` que será la sala de chat, que contendrá una lista de mensajes y un listado de los usuarios conectados a la misma. Cabe destacar que para no ser injustos y provocar falsas comparaciones debido a nuestra falta de experiencia en el desarrollo de aplicaciones en ambos frameworks, hemos tomado como base los ejemplos de chat que vienen como demostración, tanto de **IceFaces** [10] como de **APE** [11].

5.4.1. IceFaces

IceFaces es un framework que permite crear aplicaciones Web utilizando la técnica *Long polling*, actualizando la vista en el navegador Web de manera automática de modelos creados en Java. Veamos a continuación los pasos necesarios para crear un chat utilizando este framework.

Modelo El modelo para este ejemplo está realizado en Java por lo que no dista demasiado al modelo desarrollado en Smalltalk. Consta de tres clases `ChatRoom`, `Message` y `User`, cada uno conteniendo los mensajes habituales en un chat, como son el `sendMessage()`, `addUser()`, `removeUser()`, etc.

Para poder utilizar este modelo junto con **IceFaces** se deben realizar algunas modificaciones. En la clase `ChatRoom` se deben agregar dos variables de instancia y una de clase

[10] El chat además de venir entre los recursos cuando se obtiene el framework, existe un screencast explicando el desarrollo y su uso en http://www.icesoft.com/webinar/archive/dev-ajax-push-apps.html

[11] Se obtiene entre los recursos cuando se baja el framework y se puede utilizar en una demostración viva en http://www.ape-project.org/demos/1/ape-real-time-chat.html

```
public static final String ROOM_RENDERER_NAME = "
   all";
private OnDemandRenderer roomRenderer;
private RenderManager renderManager;
```

y los mensajes

```
public void setRenderManager(RenderManager
   renderManager) {
    this.renderManager = renderManager;
    roomRenderer = renderManager.
       getOnDemandRenderer(ROOM_RENDERER_NAME);
}
public RenderManager getRenderManager(){
    return renderManager;
}
```

Luego en cada mensaje del modelo donde se desee realizar una notificación para que las vistas actualicen los widgets, se debe agregar las notificaciones. Veamos el caso del método `addUser()`

```
public void addUser(User user) {
    if (!users.containsKey(user.getHandle())){
        users.put(user.getHandle(),user);
        roomRenderer.add(user);
        addMessage(participant, "joined");
        roomRenderer.requestRender();
    }
}
```

mediante el llamado `roomRenderer.requestRender()` se notifica al framework que las componentes que estén dentro del `roomRenderer` deben ser actualizados. Los mensajes `removeUser()` y `addMessage()` tendrán una estructura parecida.

En la clase `User` se debe implementar dos interfaces, la interfaz `Renderable`, para el cual se deberá definir una variable

```
private PersistentFacesState state;
```

implementar los mensajes

```
public PersistentFacesState getState() {
    return state;
}
```

```
public void renderingException(RenderingException
   renderingException) {
    chatRoom.removeUser(this);
}
```

e inicializar la variable en el constructor de la clase

```
public User() {
    state = PersistentFacesState.getInstance();
}
```

y para la interfaz `DisposableBean` se deberá implementar el mensaje

```
public void dispose() throws Exception {
    chatRoom.removeUser(this);
}
```

Vista Desde el lado de la vista Web, se debe crear una página JSP [68] en donde se creará la vista utilizando componentes provistas por el framework. Veamos el fragmento encargado de mostrar la lista de mensajes

```
<ice:panelSeries value="#{chatRoom.messages}"
                 var="message"
                 rows="10"
                 binding="#{participant.messageList}
                    ">
    <ice:outputText value="#{message}"/>
    <br/>
</ice:panelSeries>
```

Además de la página JSP, se deben crear varios archivos de configuración como son *faces-config.xml*, *web.xml* y *portlet.xml* para asociar la vista con el modelo.

Comparación Mediante la descripción de este ejemplo se pueden encontrar varias características, algunas de las cuales son similares a **Meteoroid** y otras difieren enormemente.

Puntos a favor:

- Al igual que con la capa **WebLiveWidgets** no es necesario el uso de Javascript, ni la manipulación de DOM para la actualización de los tags HTML que nos interese refrescar.

Puntos en contra:

- A diferencia de nuestro framework, el cual posee diferentes técnicas de conexión *Streaming*, **IceFaces** utiliza una conexión *Long Polling*.
- Se debe agregar código específico del framework dentro de las clases `ChatRoom` y `User` para crear las notificaciones de actualización de las componentes de **IceFaces**, lo cual atenta contra la reusabilidad. En **Meteoroid**, al aprovechar la implementación del patrón Observer sólo se escribe código exclusivo del framework en las componentes `Meteoroid`.
- La cantidad de código necesario para crear un chat es relativamente grande. Se necesitó agregar cuatro archivos (*faces-config.xml* de 35 líneas, *web.xml* de 63 líneas, *portlet.xml* de 33 líneas y *chat.jspx* de 43 líneas) para crear una página Web que funcione con el modelo, y se agregó alrededor de 10 líneas en las clases `ChatRoom` y `User`.

5.4.2. APE

APE es un framework para la creación de aplicaciones **Comet** teniendo como característica la escalabilidad respecto de la cantidad de conexiones concurrentes de usuarios que pueden usar sus aplicaciones. A continuación se describirá la implementación de un chat realizado con este framework, recalcando los aspectos buenos y malos del mismo.

Modelo Como se vio en el Capítulo 4, APE realiza la comunicación mediante canales. Es por tal motivo que no es necesario agregar ninguna línea de código relacionada con el framework dentro del modelo de dominio. Para que APE sea notificado se debe agregar algún tipo de aviso, el cual debería agregar información dentro de un canal en particular. Otra característica que provee la comunicación mediante canales es que el modelo puede estar implementado en cualquier lenguaje, y por tal motivo obviaremos la parte del modelo del chat debido a que se podría utilizar el mismo ejemplo que el implementado en Smalltalk.

Vista Del lado de la aplicación Web diseñada para utilizar la conexión Comet se deben crear tres archivos:`APE.Chat`, una página Web y un archivo de configuración en Javascript.

Para empezar se debe crear un objeto Javascript llamado `APE.Chat`, el cual hereda de `APE.Client` un objeto provisto por el framework. Éste es el encargado de conectarse al servidor **APE** para recibir y enviar información. Además, debe ser el encargado de procesar la información que llega desde el servidor y transformarla en actualizaciones Javascript. Veamos como sería escribir un mensaje al chat, en el siguiente método de `APE.Chat`

```
...
```

```
writeMessage: function(pipe, message, from){
    if(pipe.lastMsg && pipe.lastMsg.from.pubid ==
       from.pubid){
        var cnt = pipe.lastMsg.el;
    }else{
        var msg = new Element('div',{'class':'
           ape_message_container'});
        var cnt = new Element('div',{'class':'
           msg_top'}).inject(msg);
        if (from) {
            new Element('div',{'class':'ape_user','
               text':from.properties.name}).inject(
               msg,'top');
        }
        new Element('div',{'class':'msg_bot'}).
           inject(msg);
        msg.inject(pipe.els.message);
    }
    new Element('div',{
        'text':this.parseMessage(message),
        'class':'ape_message'
    }).inject(cnt);
    this.scrollMsg(pipe);
    pipe.lastMsg = {from:from,el:cnt};
    if(this.getCurrentPipe().getPubid()!=pipe.
       getPubid()){
        this.notify(pipe);
    }
}
...
```

Por otra parte se debe crear una página HTML el cual contendrá un script que inicializará el objeto `APE.Chat` asignándole un canal y un identificador

```
...
<div id="ape_master_container"></div>
<script type="text/javaScript">
    APE.Config.scripts.push(APE.Config.baseUrl+'/
       Source/Core/Session.js');
    var chat = new APE.Chat({'container':\$('
       ape_master_container')});
    chat.load({
        identifier: 'chatdemo',
```

```
            channel: 'test'
        });
    </script>
    ...
```

de esta forma, mediante el objeto `APE.Chat` se creará el chat dentro del DIV *ape_master_container*. En esta página no hay nada salvo la inicialización del objeto y el DIV contenedor debido a que todo el comportamiento se encuentra dentro del objeto `APE.Chat` y será éste quien dibuje el chat y lo actualice.

El último archivo necesario para crear la vista es un archivo Javascript de configuración (*config.js*) en donde se alojará la dirección del servidor, la URL al server donde reside el framework Javascript, los modos de trabajo y archivos que deben ser cargados, entre otras cosas.

Comparación

Puntos a favor:

- Al utilizar canales para conectarse con el modelo, es sencillo de integrar con un modelo realizado en cualquier lenguaje, siempre y cuando tenga un conector implementado para poder utilizar el canal.

Puntos en contra:

- La vista del chat Web está hecha íntegramente en Javascript, el cual es un lenguaje más de bajo nivel en comparación con un lenguaje como Smalltalk, lo que lleva a tener que codificar más para lograr resultados similares.
- Al igual que con **IceFaces** la cantidad de código necesario para crear el chat es extensa. Se deben crear tres archivos, *config.js* (7 líneas), *demo.html* (26 líneas) y *demo.js* (260 líneas).

5.4.3. Conclusión

Ambos frameworks son interesantes para el desarrollo de aplicaciones Comet, cada uno teniendo algunas buenas características y otras no tanto.

En cuanto a la codificación, realizar una vista Web de un modelo de chat con nuestro framework implica la creación de una sola componente (en nuestro ejemplo se dividió en cuatro componentes para modularizar bien el ejemplo y que sea más sencillo de entender) y un total de 171 líneas de código, pero cabe destacar que si no se cuentan los getters y setters la cantidad desciende a 134. **IceFaces** también necesita que el desarrollador codifique una cantidad similar de líneas ya que se requirió la creación de cuatro archivos (3 XML y 1 JSP) y agregar comportamiento en el modelo, dando un total 184 líneas de código. En

	APE	IceFaces	Meteoroid
Open Source	SI	SI	SI
Necesita otro Servidor Web	SI	NO	NO
Provee una API de Javascript	SI	SI	SI
Usa Streaming	SI	NO	SI
Elige la mejor técnica de conexión	SI	NO	SI
Provee widgets actualizables	NO	SI	SI
Se conecta al modelo usando	Canales	Protocolo propio	Patrón Observer
Necesita plugin en el cliente	NO	NO	NO
Es necesario saber Javascript	SI	NO	NO
Se debe utilizar archivos de configuración	SI	SI	NO
Se debe modificar el modelo	NO	SI	NO
Cantidad total de líneas para implementar el chat	293	184	171
Es reusable	NO	SI	SI

Cuadro 5.1: Comparación entre APE, IceFaces y Meteoroid

último lugar se encuentra **APE** que necesita tres componentes (1 HTML y 2 JS) y un total de 293 líneas de código. Esta gran diferencia sucede debido al nivel de abstracción que otorgan **IceFaces** y **Meteoroid** con respecto a **APE**, en el cual el desarrollador es el encargado de procesar la información y convertirla en actualizaciones de fragmentos de HTML.

Con respecto a las técnicas utilizadas por los frameworks, **APE** y **Meteoroid** comparten su interés por la elección de diferentes técnicas *Streaming* dependiendo del navegador que este utilizando el cliente, mientras que **IceFaces** utiliza siempre la técnica *Long Polling*, la cual, como fue explicado en la Sección 2.4.1, requiere de una cola de espera en el servidor para que no exista pérdida de datos.

Si observamos el Cuadro 5.1, podemos concluir que

IceFaces es atractivo desde el punto de vista de la abstracción y podría utilizarse en aplicaciones en las cuales no existe una gran tasa de actualizaciones (debido al problema que provoca *Long Polling*),

APE es mejor en aplicaciones sencillas donde no existan actualizaciones complejas ya que debe codificarlas el desarrollador utilizando puramente Javascript, que la velocidad en la respuesta al usuario sea importante (esto lo logra con la elección de la técnica de conexión) y donde la cantidad de usuarios sea extremadamente grande.

Meteoroid por su parte posee gran parte de de las buenas características de ambos frameworks, eligiendo la conexión Streaming óptima para cada navegador y proveyéndole al desarrollador un nivel de abstracción para crear actualizaciones de manera simple y siguiendo una metodología enteramente orientada a objetos.

5.5. Conclusión

A lo largo de todo el capítulo se mostró detalladamente la arquitectura de **Meteoroid**, explicando el funcionamiento de cada capa junto con sus aportes y cómo mediante la composición de las mismas se logra un nivel de abstracción cada vez mayor.

La primer capa del framework, **PushScript**, provee una *conexión Comet* realizada con la técnica óptima de acuerdo al navegador Web que el usuario esté utilizando. Además, provee un mensaje que permite enviar comandos Javascript al navegador para actualizar fragmentos de la página. Lo interesante de esta capa es que de manera muy simple se puede enviar información al cliente, lo malo es el nivel de abstracción que el desarrollador posee para actualizar, debido a la necesidad de codificar en Javascript para realizar las modificaciones y teniendo que manejar por su cuenta las dependencias que quiera tener con el modelo. Esta capa puede ser de mucha utilidad en aplicaciones sencillas, que requieran actualizaciones muy simples o que se requiera tener más control sobre el Javascript que se quiera enviar.

Sobre la primer capa se construyó una segunda, llamada **Observer**, con la intención de abstraerse de la *conexión Comet* y el uso de Javascript. Acercarndo cada vez más, a un nivel de abstracción que se asemeja más al lenguaje en el que se decidió desarrollar las aplicaciones, Smalltalk. Por este motivo se creó una serie de mensajes que permiten la automatización de actualizaciones, teniendo en cuenta algún evento asociado, codificando todo esto en Smalltalk y sin la necesidad de preocuparse en el manejo de dependencias. Esta capa resulta ser muy interesante con respecto al nivel de abstracción que proporciona y siendo extremadamente sencilla de utilizar.

La última capa de **Meteoroid**, **Web Live Widgets**, culmina con el nivel máximo de abstracción permitiendo crear aplicaciones Comet siguiendo la misma metodología que se utiliza en VisualWorks para el desarrollo de aplicaciones de escritorio. El framework provee una serie de widgets, asociados con aspectos del modelo, que son actualizados automáticamente por el framework cuando sea necesario. Esta capa es muy interesante ya que permite a los desarrolladores ocuparse sólo de los problemas de su aplicación y no en los aspectos más técnicos de cómo mostrar la información en sus páginas de manera actualizada o que el HTML se vea igual en todos los navegadores Web.

Por último se realizó una comparación con otros frameworks mostrando las ventajas que posee **Meteoroid** en cuanto a la facilidad de uso y codificación, siendo el que menos líneas de código necesitó codificar un ejemplo de chat, llegando en el caso de **APE** hasta un 40 % menos de líneas.

A lo largo de este capítulo se mostró cómo se pudo lograr el desarrollo de un framework que permite, de manera simple y orientada a objetos, la creación de aplicaciones Comet, fomentando al desarrollador a centrarse en los problemas de su aplicación y no tener en cuenta la *conexión Comet*, ni el uso de Javascript ni

el manejo de dependencias.

Capítulo 6

Conclusión y Trabajo a Futuro

En el presente trabajo de grado se ha diseñado una arquitectura flexible para el desarrollo de aplicaciones Web vivas. Como pruebas de concepto se han implementado diversas aplicaciones, se probó el framework en ambientes reales (aunque reducidos) y se mostró las diferentes formas de poder usarlo. Por último, se comparó una aplicación con otros frameworks para ver el desempeño de nuestra arquitectura. En este capítulo se comentarán las conclusiones obtenidas a lo largo de la tesis y se describirán los trabajos que se puedan realizar en un futuro.

6.1. Conclusión

Desde que el término Comet fue acuñado en el año 2006 por Alex Russell, diversas herramientas fueron surgiendo para tratar de unificar el concepto novedoso. Cada una trató de dar algún tipo de soporte que la anterior no tomaba en cuenta, ya sea en abstracción o en la forma de generar un producto cometificado más rápido y sencillo. En respuesta a esta motivación de generar nuevos frameworks surgió de forma natural el nuevo *futuro* estándar: HTML5. Comet todavía es una solución interesante, pero es cierto que es temporal y por tal motivo los frameworks actuales sólo buscan generar una manera de conectar al servidor con el cliente de la forma que sea posible, sin importar si es de hecho una *conexión Comet* o un simple *long polling*.

La motivación que fomentó este trabajo fue inicialmente un *medio* para poder conseguir desarrollar aplicaciones Web en dispositivos móviles que rompan con la barrera que impone HTTP. Dada la naturaleza de Comet y la posibilidad de uso real que existía en **Meteoroid**, se tomó como alternativa que deje de ser un *medio* para otro proyecto para pasar a ser un *fin*.

En los primeros capítulos se comentó el estado del arte de las diversas herramientas comerciales con Comet (independientemente de sus lenguajes de apli-

cación), junto con un poco de historia para entender cómo las páginas Web evolucionaron a algo mucho más robusto como lo que son hoy en día. En este trabajo de grado se presentó una arquitectura que permite dividir el modelo de su representación Web visual. Para ello se introdujo en una primera instancia las metodologías para conectar un servidor con un cliente vía una conexión con streaming, los detalles de cómo es posible embeber componentes `Meteoroid` y un ejemplo de cómo manipular el navegador con el uso de Javascript. Luego se continuó con el desarrollo de la segunda capa que provee una forma de, a partir de un modelo, actualizar vistas escribiendo enteramente en código Seaside. Finalizando con componentes Web que entienden cómo y cuándo dibujarse, nuevamente, a partir de un modelo.

Lo interesante es que cada capa tiene características disjuntas, donde cada una de ellas puede ser usada por una aplicación al mismo tiempo ya que no son excluyentes. Esto permite que en una misma aplicación Web se puedan utilizar *live widgets* de la capa **WebLiveWidgets**, junto con **PushScript** para manipular alguna librería Javasctipt para gráficos de barra (un ejemplo de esto se observa en el ejemplo *Subasta*).

Para poder poner a prueba nuestro trabajo, se realizó diferentes tests y comprobaciones sobre servidores que están actualmente funcionando [1], así como también se realizaron comparaciones contra otros dos frameworks para verificar cuán sencillo es desarrollar con nuestras capas. Los resultados fueron bastante prometedores, tanto en el servidor que se mantiene corriendo desde hace meses, hasta las comparaciones con **ICEFaces** y **APE** donde los resultados arrojan números muy positivos a nuestra arquitectura. Lo importante del último punto, es que ambas alternativas son elementos comerciales que actualmente se usan en diferentes contextos, frente a un trabajo que se desarrolló en un ambiente exclusivo de investigación.

6.2. Trabajo a Futuro

El área de Comet tiene un alcance grande, y por tales motivos no puede ser abarcado de forma completa dentro de este trabajo de grado. A partir del trabajo realizado se ha identificado el siguiente trabajo a futuro:

Herramientas de debugging Programar en Smalltak trae una inmensa ventaja a la hora de debuggear código. **Meteoroid** se acopla perfectamente a estos debuggers, pero sería interesante tener un monitor de `Handlers` para saber el estado actual de cada componente. Como complemento sería bueno tener otro monitor encargado de ver las dependencias entre los *live widgets* y los diferentes modelos de forma tal de poder estudiar los diversos problemas que pueden aparecer.

[1] URL: http://cag.lifia.info.unlp.edu.ar:443/Meteoroid

Estudio de performance y escalabilidad Dado que cuando concebimos **Meteoroid** nos focalizamos en que sea una herramienta fácil de instanciar y cómoda en su uso, no nos concentramos en las características optimizables del mismo. Terminada la etapa de diseño es ahora necesario volcarnos al estudio exhaustivo de cantidad de conexiones posibles, cómo se comporta el servidor, etc.

Extender Live Widgets La capa **WebLiveWidgets** del framework está pensada para que sea extensible por cualquier usuario de **Meteoroid**, para esto tratamos de extraer tanto comportamiento como sea posible de los *live widgets* de forma tal que extenderlo sea natural y simple. Lo óptimo sería que la suite de widgets contuviera tantos como sea posible, como son los sliders, spins buttons, tabs, tree view, etc.

Creación de herramientas para desarrollar GUIs Web En VisualWorks existe un editor de vistas llamado *GUI Painter Tool* [93], el cual define un método (spec) que es llamado cada vez que la vista es creada. Este spec posee todos los elementos de la vista (básicamente una composición de widgets), a los cuales los va a ir instanciando y colocando en la ventana creada. Esta aproximación no dista mucho de cómo podrían ser creadas las componentes **Meteoroid** ya que los *live widgets* comparten el mismo comportamiento que los widgets estándar de VisualWorks. Por lo tanto podría utilizarse ese spec para crear interfaces Web automáticamente a partir de una de escritorio ya creada.

Extender casos de prueba A pesar de que se han realizado diversos TestCases, se deben actualizar gran parte de ellos y escribir otros para la capa **WebLiveWidgets**. Nosotros consideramos que los tests son parte fundamental a la hora de programar, y más aún cuando hablamos de un framework, ya que los test ayudan a mantener consistencia entre cambios. El objetivo sería escribir una batería de tests lo suficientemente grande como para corroborar todos los aspectos que son abordados por **Meteoroid**.

Unificación de tecnologías Un veta interesante sería ver cómo hacer que nuestro trabajo no solo se comporte en algunos aspectos como una aplicación de escritorio, sino que sienta por completo como tal. La forma de lograr esto sería uniendo tecnologías, CSS más Javascript que emulen los widgets nativos del sistema operativo, más la utilización de un software como Prism [94] para que el usuario no vea los botones del navegador, ni la barra de URL, etc. Esto permitiría desarrollar una vista mucho más moderna y práctica, que los actuales widgets que provee VisualWorks (los cuales son prácticamente los mismos desde hace más de 10 años).

Portar Meteoroid Todo nuestro trabajo fue realizado en VisualWorks, un sabor de Smalltalk muy interesante dado que entre Smalltalks es el más rápido

debido a su máquina virtual. Nuestro objetivo es portar **Meteoroid** a diferentes sabores de Smalltalk, pero con especial detalle en Pharo. Esto nos parece óptimo porque Pharo (entre otros Smalltalks) es una distribución con licencia 100 % MIT [95], y creemos que mantener el trabajo usable a sistemas completamente MIT y Open Source es uno de los objetivos de la investigación: compartir conocimiento. Dejar andando **Meteoroid** sólo para un sistema como VisualWorks, el cual es pago, no nos parece lo mejor.

Transformadores de valores Los *live widgets* son elementos muy cómodos a la hora de crear vistas dependientes de un aspecto de un modelo. El problema que tienen nuestros widgets es la ausencia de configuración, de forma tal que si el modelo tiene un objeto de tipo `Date`, su `WebLiveWidget` lea y escriba en formato `Date`. Lo mismo ocurre con los números, caracteres, etc., donde es necesario de otro objeto que se encargue de mapear los strings que vienen del cliente en objetos de primer orden en Smalltalk. Esto se puede lograr usando la clase `PrintConverter`, la cual es usada por los widgets en las aplicaciones de escritorio de VisualWorks. Este objeto es instanciado para convertir a números, fechas, etc. y se asocia a un widget. Luego, si fue instanciado para una fecha por ejemplo, convierte la fecha a string para mostrarla en la vista y transforma el string a una fecha cuando se desea modificar el modelo.

Mejorar conexiones A pesar que el presente trabajo ya cuenta con un buen manejo de conexiones, es posible mejorarlo más. Sería interesante proveer un tipo de conexión WebSocket para navegadores que ya soporten el Draft de HTML5, así como también elegir si se desea otra técnica de Comet en diferentes aplicaciones Web. También sería bueno proveer soporte para reconectar a los clientes Web, ya que hoy en día es probable que algún usuario se encuentre utilizando una red inalámbrica en vez de cableada, llevando eso a posibles desconexiones.

Apéndice A

Seaside

Cuando uno habla de aplicaciones Web tiene que pensar de forma casi instantánea de algún framework que acompañe su desarrollo, ya que implementar un sistema con grandes requerimientos (manejo de sesiones, estados, clientes, modelo de dominio, base de datos, etc.) en lenguajes como PHP [96] puro, CGI [97] no parece posible. Existen muchos frameworks desarrollados en diferentes lenguajes para atacar este problema de abstraerse del protocolo y dedicarse a desarrollar de forma completa una solución al problema, pero Seaside posee algunas buenas características que merecen ser explicadas.

Este apéndice tiene como objetivo ilustrar las diferentes capacidades que tiene Seaside, un framework para desarrollar aplicaciones Web puramente en Smalltalk. Se desarrollarán los puntos fuertes de esta herramienta, y se mostrará como generar diferentes sitios con la misma, finalizando con un ejemplo de aplicación. Es importante destacar que Seaside es una herramienta Open Source, con una creciente comunidad y hasta se le puede dar el crédito de ser el motivo por el que las personas empiezan a retomar un interés por Smalltalk.

A.1. Introducción

Seaside es un framework Open Source escrito en Smalltalk el cual basa la construcción de aplicaciones Web en el uso y reuso de componentes y continuations. Los desarrolladores sólo deben extender clases del framework para poder crear aplicaciones Web dinámicas, con excelente manejo de control flow (flujo de trabajo) [2, página 4], buena interacción con el usuario y alta reusabilidad. En algunos aspectos, realizar una aplicación con Seaside se asemeja mucho con la realización de una aplicación de escritorio, donde existen vistas y subvistas para renderizar diferentes campos de datos. En particular las vistas/subvistas de Seaside son elementos llamados componentes (esto se explicará con más detalle a

continuación).

Seaside fue desarrollado por Avi Bryant and Julian Fitzell en el año 2002 bajo la versión 0.9, con el objetivo de mejorar sus herramientas de trabajo en su consultora. A fines del 2003 surgió la versión 2.0, donde se habían refactorizado ciertos elementos del framework y eliminado otros (como por ejemplo los templates). Seaside fue desarrollado en sus inicios en un entorno Squeak [98][99], pero luego fue portado a otros dialectos Smalltalk como: Dolphin [100], Cincom VisualWorks, etc. Desde el año 2009 el desarrollo de Seaside se mantiene en Pharo [101][102], un fork de Squeak, liderado por los core-developers Julian Fitzell, Lukas Renggli y Philippe Marschall.

A.2. Características

A pesar de que Seaside es una excelente herramienta para la construcción de aplicaciones Web, hay dos aspectos que no contempla de forma "estandarizada", no hay REST (Representational State Transfer) URLs [103] ni tampoco manejo de templates. No existe REST por defecto en Seaside debido a que utiliza las URLs en otra forma. Las URLs albergan información relevante al usuario y a su estado de navegación (más adelante se explicará esto junto con las continuations). Por este último motivo, las URLs en Seaside son generadas de forma dinámica (el desarrollador no tiene forma de manipularlas a gusto). Con respecto a los templates, Seaside los aparta debido a que la lógica de los mismos es la de separar la responsabilidad de cómo se describe una página en un archivo y cómo se ve en otro. Esta separación provoca que se generen muchos archivos para poder maquetar e ilustrar una aplicación Web determinada. Seaside promueve esta separación, pero no a través de un motor de templates, sino con la separación y encapsulamiento de componentes (que serán detallados en su sección). Cabe destacar que a pesar de que Seaside no implementa estas dos funcionalidades, existen formas de realizarlas. En este apéndice no se discutirán tales temas, porque escapan al interés general de esta tesis, pero nos parece importante destacar que es una elección de Seaside no proveerlas y no una imposibilidad.

Seaside, no se preocupa por los dos aspectos detallados anteriormente, porque a través de la remoción de los mismos pudo encontrar otras características mucho más interesantes y útiles que tener REST URLs y el uso de templates. Las ganancias que tiene este framework hacen que el desarrollo de aplicaciones Web parezca simple, dada la vasta funcionalidad que provee. Entre ellas, se pueden destacar:

- Component: unidad mínima que refleja parcial o totalmente la vista del modelo de negocios.

- Callbacks: acciones a realizar sobre eventos del cliente en el servidor.

- Tasks: permiten definir el workflow de la aplicación.
- Ajax: posibilidad de crear elementos Web 2.0 [104] de forma sencilla.
- Hot debugging y Recompilation: arreglar bugs sin necesidad de tener que parar al servidor y levantarlo.

A continuación se detallarán cada una de las características nombradas anteriormente.

A.2.1. Component

Los elementos principales en las aplicaciones de Seaside son los componentes, objetos que se encargan de definir y dibujar la vista. Los mismos deben ser vistos como la vista y controlador del modelo, según el patrón MVC (ver Apéndice B). Cuando se desarrolla un componente, hay que subclasificarlo de `WAComponent` para que herede el comportamiento de la clase abstracta, y reimplementar el método `#renderContentOn:`. La clase abstracta `WAComponent` implementa el Template Method [75, Template Method pattern], donde el hook (mensaje obligatorio que las subclases deben implementar) es el `#renderContentOn:`. Gracias a este mensaje, en la etapa de dibujado, se invocará al componente y se ejecutará el mensaje `#renderContentOn:` para que dicho componente se pueda dibujar, donde el parámetro es un objeto `WAHtmlCanvas`. Este objeto entiende diferentes mensajes para crear de forma conveniente y práctica los tags HTML. Es importante mencionar que por diseño Seaside solo genera XHTML, ya que cada tag es un objeto que enmascara al resto, y por lo tanto en el ciclo de vida de una componente existe tanto el inicio y aperturas de esos tags, como así en la finalización de dicho ciclo se encuentran los diferentes cerrados de los diferentes tags.

Los componentes son también la unidad mínima de estado que existe en Seaside, representados por las variables de instancia, permiten definir módulos encastrables. Cada componente se responsabiliza de escribir en su `#renderContentOn:` los elementos necesarios para poder manipular los estados, de forma tal que luego sean accedidos.

Para dibujar el siguiente fragmento de HTML

```
<html>
    <body>
        <h1>Ejemplo</h1>
        <div id='ejemploID'>
            Hola mundo
        </div>
    </body>
</html>
```

es necesario escribir el siguiente código Seaside

```
>>renderContentOn: html
html heading level: 1;
     with: 'Ejemplo'.
html div id: 'ejemploID';
     with: 'Hola mundo'.
```

Las componentes permiten dividir un sitio en diferentes módulos, da la posibilidad que puedan ser reutilizados en diversas aplicaciones, como puede ser el caso de una componente encargada del manejo del login que puede ser implementada una sola vez y ser reutilizada en múltiples aplicaciones. Por otro lado también es posible la composición de componentes, que permite la creación de otras más complejas mediante la composición de componentes más pequeñas, pudiendo de esta forma separar una vista compleja en varias más sencillas. Por ejemplo, como se puede observar en la Figura A.1, se puede crear una aplicación compuesta por varias subcomponentes. En este caso, se creó una aplicación que permite manejar el Store [105] de VisualWorks de forma Web separando en varias componentes, cada una encargándose de un problema en particular. Esta creada por una componente llamada `StoreManagement` que a su vez esta compuesta por un menú (`StoreManagementNavigation`) que contendrá muchas componentes (una por cada entrada del menú), y que en el caso de la figura mostrará la componente `StoreRepositoryConnection`, que contiene la lógica necesaria para mostrar y conectar las conexiones con el Store.

Notar que como se comentó anteriormente, si en otra aplicación se necesitara conectarse al Store, se podría incluir la componente `StoreRepositoryConnection` sin ningún problema.

A.2.2. Callbacks

Los callbacks se pueden definir como acciones a tomar del lado el servidor frente a eventos ocurridos en los navegadores Web. Con los componentes tenemos una forma de mostrar información a los clientes, pero esa estructura no define como el usuario puede modificar los estados de los mismos. Los callbacks son una suerte de acceso remoto, donde mediante bloques-Smalltalk [99, página 57] de Smalltalk se realiza una acción en el servidor (quizás es comparable este mecanismo con las funciones anónimas [106, página 99]). Gracias a los callbacks, es posible modificar el modelo residente en el servidor de una forma sencilla. En particular, los buttons y anchors requieren un bloque con cero parámetros, pero el resto de los tags HTML como los input fields, check boxes, etc. sí toman parámetros dado que son elementos que pertenecen usualmente a formularios FORM de HTML. El parámetro en cuestión, es el elemento al que el callback está vinculado, si vemos el ejemplo a continuación

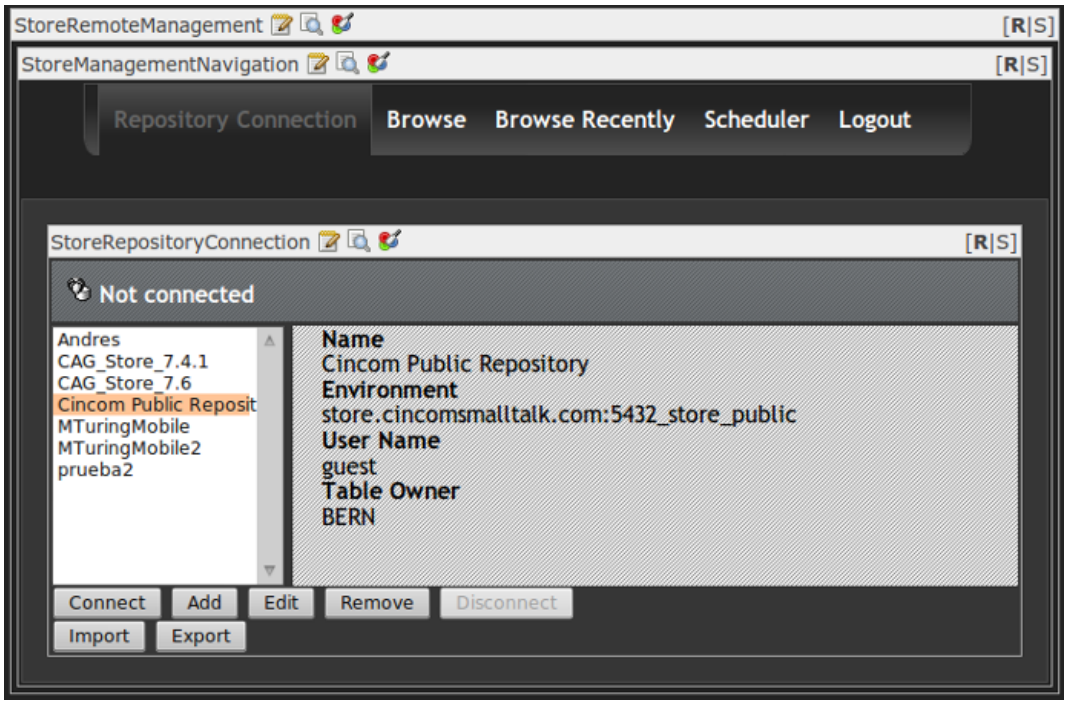

Figura A.1: Composición de componentes

```
>>renderContentOn: html
html anchor
    with: 'Imprimir hora en transcript';
    callback: [self log:
        TimeStamp now printString].
```

generará el HTML necesario para que cuando el usuario haga click en el vínculo, se imprima la hora actual en el Transcript (consola) [99, página 11] de la imagen de VisualWorks que se esta corriendo en el servidor, como se observa en la Figura A.2.

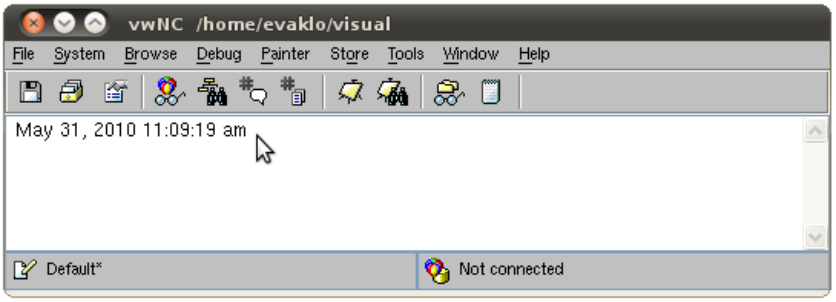

Figura A.2: Captura de Transcript de VisualWorks

Un ejemplo de un callback más interesante, sería con un SELECT. Supongamos que se desea mostrar una lista de usuarios y que al hacer click en un botón se mostrará el usuario seleccionado en el Transcript

```
>>renderContentOn: html
html form:[
  html select
        list: self users;
        callback: [ :user |
            Transcript show: 'El usuario elegido
               es: ', user name
        ].
  html submitButton.
]
```

y si suponemos que se elige el usuario "Julian Gomez", en el servidor obtendremos la siguiente salida luego de que el usuario presione el botón de submit de la página, como se observa en la Figura A.3.

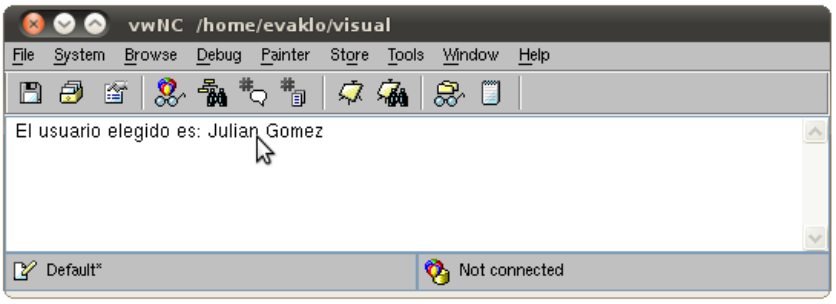

Figura A.3: Captura de Transcript luego de imprimir el usuario

lo importante en este segundo ejemplo radica en cómo se manipulan los datos del cliente: en muchos tags, como es el caso del SELECT, RADIOBUTTON, CHECKBOX, etc., no hace falta parsear datos ni tampoco hace falta procesarlos a partir del requerimiento del usuario [1], solo se acceden y se utilizan los datos, y luego Seaside se encarga del resto. Como se ve en el ejemplo descrito, el parámetro del callback es un objeto de la colección de usuarios, y no un nombre que represente al mismo. Los callbacks proveen un nivel de abstracción que rompe con el esquema tradicional de los formularios en HTML, donde hay que tener cuidado con los choques de nombres, no repetir grupos, o hasta ver cómo parsear una cantidad importante de datos, dependiendo de cada formulario.

[1] En PHP se realiza mediante $_GET["aParameter"]. En Struts request.getParameter("aParameter")

A.2.3. Tasks

Los componentes sirven para visualizar datos y encapsular estados, pero no están estrictamente diseñados para manipular el control de la aplicación en composición con otros componentes diferentes. La idea general de programación en Seaside es la misma que se espera en un lenguaje orientado a objetos, modularizar y separar, cuanto más se pueda, comportamiento en diferentes objetos. El "pegamento" necesario para que Seaside permita construir buenas aplicaciones Web son las Tasks, un tipo especial de componentes las cuales no dibujan XHTML directamente, sino de manipular el control flujo de la aplicación. Las tareas tienen como propósito ordenar la secuencia lógica de cuales componentes deben ser visualizadas en un momento dado, y delegar el dibujado del XHTML a las componentes que va a llamar.

Una tarea, subclase de `WATask`, no debe redefinir el método `#renderContentOn:` sino el método `#go`, el cual es invocado inmediatamente ni bien la tarea comienza a mostrarse. Veamos un ejemplo definiendo la tarea `Sum2NumsTask` la cual pide por dos valores numéricos para luego ser mostrado el resultado de su suma

```
Sum2NumsTask >>go
| num1 num2 |
num1 := (self request: 'Ingrese un numero')
   asNumber.
num2 := (self request: 'Ingrese un segundo numero')
    asNumber.
self inform: 'La suma de los numeros es: ' ,
    (num1 + num2) printString
```

los mensajes `#request:` e `#inform:` son mensajes provistos por Seaside, los cuales son dos componentes que se encargan de tomar un valor del usuario (`#request:` componente `WAInputDialog`) y de mostrar un mensaje al usuario (`#inform:` componente `WAFormDialog`). Con ese simple mecanismo, hemos podido construir una aplicación que toma dos números para luego mostrar la suma al usuario.

Las tareas son importantes para definir el flujo de una aplicación en Seaside, proveen aislamiento de cómo dibujar el XHTML y se focaliza en las condiciones que deben darse para llamar a una componente u otra, dependiendo de lo que se defina en el `#go`.

A.2.4. Ajax

Seaside provee una buena integración con diversas herramientas para la manipulación de Ajax en las aplicaciones Web. El uso de Ajax no solo es conveniente

para aliviar la carga en el servidor, dado que sin esta tecnología actualizar un cambio en el cliente implicaría tener que procesar toda la página nuevamente, sino que además proporciona actualizaciones visuales en el cliente sin forzar que el usuario haga click en un link o un botón.

Seaside mapeó una librería de Javascript llamada Prototype, la cual ayuda a la construcción de Javascript en el lado del cliente. Prototype fue creado para mejorar la usabilidad de requerimientos Ajax y manipulación DOM, abstrayéndose de los diferentes problemas subyacentes a Javascript (como por ejemplo, incompatibilidades entre diferentes navegadores Web). Además de esta librería existe otra llamada script.aculo.us, la cual está construida encima de Prototype y se encarga de realizar diferentes mejoras a nivel de gráficas y efectos Javascript. Esta última, también se encuentra mapeada en Seaside.

El mapeo de ambas librerías es una completa integración bajo el paquete Smalltalk Scriptaculous, el cual provee toda las funcionalidades de Prototype y script.aculo.us a través de objetos Smalltalk. Cada clase Javascript, de cualquiera de los dos frameworks, tiene una clase que representa lo mismo en el mundo de Smalltalk, permitiendo a los desarrolladores escribir código Smalltalk que luego generará snippets (porciones de código) Javascripts embebidos en el XHTML. Veamos un ejemplo de como realizar un actualizador de información en una componente

```
Reloj>>renderContentOn: html
html div
    script: (html scriptaculous periodical
        frequency: 1 second;
        callback: [ :ajaxHtml | ajaxHtml render:
           Time now ]);
    with: Time now
```

producirá el código necesario para visualizar un reloj que irá actualizando el tiempo cada un segundo. Notar que no hace falta escribir ningún tipo de URL, ni de Javascript, solo código Smalltalk. El bloque del callback será evaluado cada 1 segundo en el servidor debido a que cada 1 segundo desde Javascript, en el cliente, se realizará un llamado Ajax al mismo.

A.2.5. Hot debug y Recompilación

La mayoría de los frameworks para el desarrollo de aplicaciones Web proveen un soporte limitado para debuggear, donde usualmente en caso de falla se tiene un número de línea y un stack trace (log donde se muestra una cadena de ejecución hasta que finaliza la ejecución por un problema) para poder encontrar y resolver la falla ocurrida. Usualmente para debuggear este tipo de aplicaciones, se usan

salidas a consola a lo largo del código, lo que puede llevar a ensuciar el código solo para ver en dónde está en efecto la problemática.

La forma de codificar en Smalltalk es de naturaleza incremental e interactiva. Los desarrolladores pueden agregar, editar e inclusive borrar código en una aplicación de Smalltalk mientras la misma se está ejecutando. Las excepciones en Smalltalk son objetos de primer orden que referencian al contexto original donde fueron levantadas, y no son manejadas (siguen vivas) hasta que algún manejador de forma explícita decide qué hacer con ellas.

En un contexto Web este manejo de excepciones sería extremadamente útil, dado que permitiría editar una aplicación Web sin la necesidad de parar la aplicación, matar al servidor, encontrar y reparar el error, levantar el servidor para luego probar nuevamente. Seaside, provee un mecanismo en conjunción con Smalltalk para poder reparar un error y ver los resultados en el mismo contexto donde se produjo.

Apéndice B

Desarrollo de aplicaciones de escritorio

Entender el desarrollo de aplicaciones de escritorio en un sistema como VisualWorks es muy interesante debido a que se realiza siguiendo patrones de diseño y una metodología orientada a objetos puro. Meteoroid se basa en esta misma metodología pero llevándola al desarrollo de aplicaciones web, y por lo tanto una vez entendido el desarrollo de aplicaciones de escritorio, el paso a la web es trivial.

En este apéndice se describirán los patrones MVC y Observer, que servirán para entender la metodología utilizada para crear las aplicaciones. Luego se explicará una implementación particular del patrón Observer, llamada Announcements, que es utilizado en el framework Meteoroid. Una vez terminado, se explicarán los `ValueModel` y como estos hacen que la creación de aplicaciones utilizando MVC pueda ser realizado de una manera simple y efectiva. Por último, se explicará la implementación de una aplicación para demostrar la facilidad de desarrollo que ofrece esta metodología.

B.1. MVC

A la hora de construir aplicaciones interactivas, la modularidad de los componentes provee grandes beneficios. La modularización de las unidades funcionales hace que sea más fácil para el diseñador de la aplicación entender y modificar cada unidad en particular, sin tener que saber acerca del resto de la aplicación. Esta modularización puede ser realizada en diversos niveles, tanto a niveles más bajos (dentro de un modelo), como a niveles más generales (por ejemplo a la hora de crear las vistas de la aplicación). Dentro de este último caso, se desarrolló un patrón para colaborar en la modularización de las aplicaciones en tres componentes. Esta división separa por un lado las partes que representan el modelo

del dominio de aplicación subyacente, por el otro la manera en que el modelo se presenta para el usuario y por último la manera en que el usuario interactúa con el modelo.

El patrón Model-View-Controller (MVC) es la refactorización de la aplicación en tres capas, en la cual, algunas clases deben hacerse cargo de las operaciones relacionadas con el dominio de aplicación (el modelo), otras de la visualización del estado de la aplicación (la vista), y otras se encargan de la interacción del usuario con el modelo y la vista (el controlador).

Modelo. El modelo de una aplicación es la implementación de la estructura central de una aplicación. Esto puede ser tan simple como un número o el modelo de un contador, o más complicado, como un objeto complejo del estilo de un objeto que represente a una persona.

Vista. Las vistas son las encargadas de manejar gran parte de lo referido al entorno gráfico, que requieren información del modelo para luego mostrarlo. Las vistas no sólo contienen los componentes necesarios para la visualización, sino también pueden contener subvistas y estar contenidas en otras vistas.

Controlador. Los controladores sirven de unión entre los modelos asociados, las vistas y los dispositivos de entrada (teclado, mouse, etc.). Los controladores también deben planificar las interacciones con otras vistas/controladores.

En el esquema descrito anteriormente, las vistas y los controladores están estrechamente relacionados y tienen interés en exactamente un modelo, pero ese modelo puede tener una o varias vistas y los controladores asociados a él. Por este motivo, para maximizar la encapsulación de datos y mejorar la reutilización de código, las vistas y controladores conocen a su modelo de manera explícita, pero los modelos no deben conocer explícitamente a sus vistas y controladores.

Un cambio en un modelo es a menudo desencadenado por un controlador, que refleja una acción del usuario en un mensaje enviado al modelo. Este cambio debe reflejarse en todas sus vistas, no sólo a la vista asociada con el controlador que inició el cambio. Para manejar la notificación de cambios, fue necesario crear la noción de dependencia de objetos. Las vistas y los controladores de un modelo son registrados en una lista como dependientes del modelo, para ser informados cada vez que se cambia algún aspecto del mismo. Cuando un modelo cambia, se envía un mensaje de notificación a todos sus dependientes, y luego cada vista o controlador responde a los cambios del modelo en forma apropiada. A continuación veremos con mayor detalle un patrón de diseño que permite representar esta noción de dependencia necesaria.

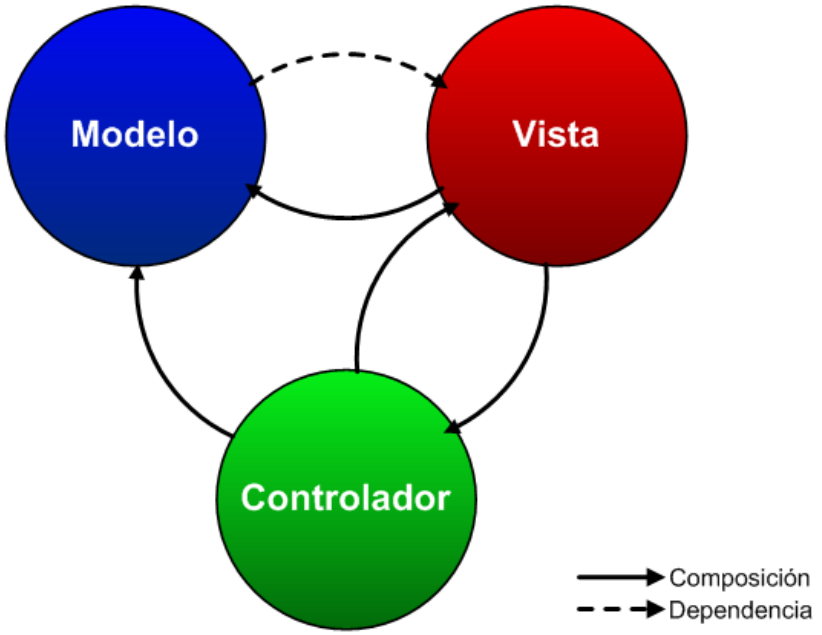

Figura B.1: Patrón MVC

B.2. Patrón Observer

Como se explicó en la sección anterior, al dividir un sistema en un conjunto de objetos que se relacionan entre sí, es necesario mantener toda la información actualizada. Pero a veces es deseable que todos estos objetos relacionadas no se encuentren estrechamente acoplados debido a que imposibilita la reutilización. Un ejemplo muy claro de este problema es el planteado en la relación entre las vistas y el modelo, ya que pueden existir diferentes vistas sobre un modelo, las cuales pueden estar interesadas en diferentes aspectos. Si el modelo estuviera estrechamente relacionado con una vista haría que su reutilización sea muy dificultosa.

Para aclarar el problema planteado, consideremos una clase `Clock` y una vista, `ClockView`, que dibuja un reloj analógico. La vista posee una variable `model` que contendrá la instancia del reloj, y dicha instancia también conocerá a su vista mediante una variable `view`, para que puedan notificarse los cambios. Por lo tanto en la inicialización de la vista existirá el siguiente fragmento de código

```
ClockView>>initialize: aModel
    self model: aModel.
    self model view: self.
    ...
```

De esta forma ambos se conocen explícitamente y cada vez que la instancia

de `Clock` realice un cambio notificará a su vista que debe actualizarse mediante el mensaje `#redisplay`

```
Clock>>tick
    self time addSeconds:1.
    self view redisplay.
```

Ahora supongamos que deseo crear otra vista sobre el mismo modelo pero que muestre la información en un reloj digital. `ClockView` pasaría a ser una clase abstracta con dos subclases, `AnalogClockView` y `DigitalClockView`, las cuales mostrarán la hora de forma diferente. La variable `model` servirá en las vistas, ya que cada instancia tendrá la propia, pero el modelo poseía una variable que alojaba a una sola vista. Por lo tanto, debemos agregar otra variable para alojar a la segunda vista, por ejemplo una variable analogClockView y otra digitalClockView, lo que generará que el código sea

```
Clock>>tick
    self time addSeconds:1.
    self analogClockView redisplay.
    self digitalClockView redisplay.
```

En caso de existir otra vista, tendremos otra variable más. Esto seguirá ocurriendo por cada vista que se desee tener sobre un modelo, y claramente se observa el motivo por el cual conocer explícitamente a sus vistas no escala.

Por este motivo se desarrolló el patrón Observer [75, Observer pattern], el cual es un patrón de diseño que describe cómo establecer estas relaciones. Los objetos clave en este patrón son el *sujeto* y el *observador*. Un *sujeto* puede tener un número ilimitado de *observadores* interesados en algún aspecto. Todos los *observadores* son notificados cada vez que el *sujeto* experimenta un cambio en su estado. A raíz de esta notificación, cada *observador* consultará al *sujeto* para sincronizar su estado con el del *sujeto*.

Como se puede observar en la Figura B.2, este patrón consta de cuatro componentes:

Sujeto abstracto. Esta clase abstracta es quien conoce a sus *observadores* y provee un protocolo por el cual un objeto que se encuentra interesado puede crear una dependencia, o bien deshacer esta dependencia cuando ya no le interese.

Observador abstracto. Esta clase define un protocolo de actualización para los objetos que deben ser informados de los cambios de los objetos de los cuales se hizo dependiente.

Sujeto concreto. Esta clase es quien guarda la relación con los observadores concretos, para enviarles notificaciones cuando cambia su estado.

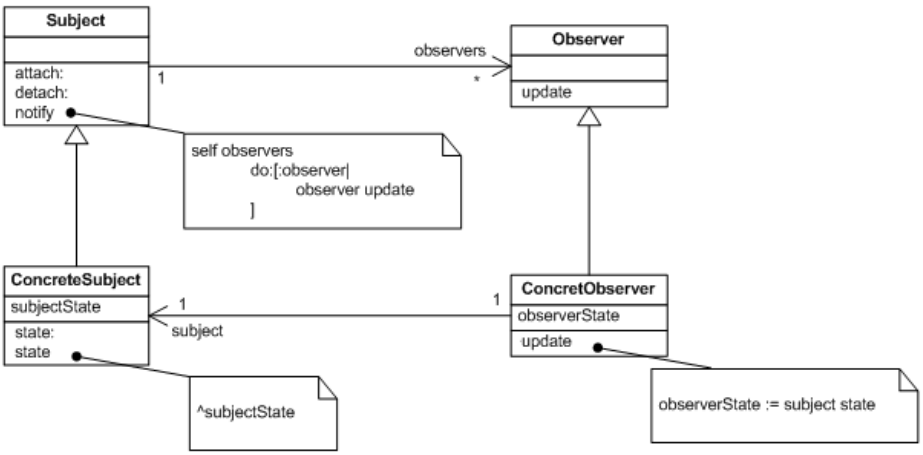

Figura B.2: Patrón Observer

Observador concreto. Este objeto tiene una referencia al sujeto concreto por el cual está interesado e implementa el protocolo provisto por el observador abstracto para poder actualizar su información.

El patrón Observer permite modificar tanto los *sujetos* como los *observadores* independientemente, permitiendo reusar los *sujetos* sin reutilizar a sus *observadores*, y viceversa. A su vez permite agregar *observadores* sin modificar el *sujeto* u otros *observadores*.

Este patrón fue el que hizo posible que el patrón MVC fuera implementado en VisualWorks de una forma simple y extensible. El modelo de MVC representa al *sujeto*, mientras que las vistas simbolizan los *observadores*. De esta forma existe una independencia entre el modelo y las vistas. Para que este patrón pueda ser utilizado por cualquier modelo hecho en este lenguaje, se proporciona un mecanismo de dependencia general, poniendo los protocolos del *sujeto* y del *observador* en la clase padre de todas las demás clases en el sistema (Object). Esta clase provee un protocolo compuesto por los mensajes `#addDependent:` y `#removeDependent:`, que permiten agregar y eliminar dependientes. Para mantener la simplicidad, los cambios son notificados mediante el mensaje `#change: aSymbol`, el cual notifica a todos los dependientes del objeto que ejecuta el mensaje mediante el llamado al mensaje `#update: aSymbol`. Por lo tanto todos los objetos dependientes deberán reimplementar el mensaje `#update:` en caso de querer realizar una acción cuando ha sucedido algún cambio en particular. El símbolo enviado como argumento colabora para poder diferenciar 2 eventos distintos que pueda llegar a recibir un objeto.

Retomando el ejemplo de los relojes, consideremos nuevamente las vistas `AnalogClockView` y `DigitalClockView`. Ahora el modelo en vez de tener sus vistas explícitamente en variables para notificarles sus cambios, utilizará el pa-

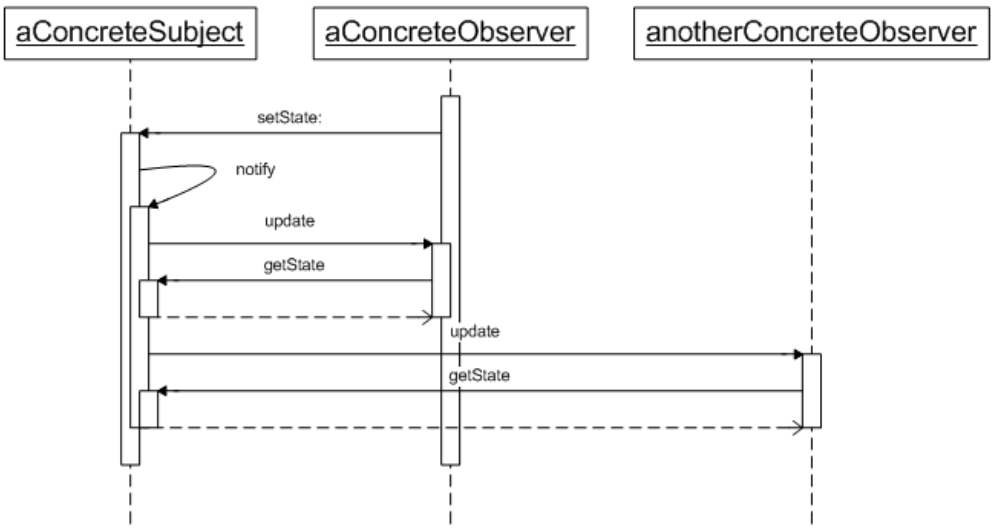

Figura B.3: Diagrama de secuencia de Patrón Observer

trón Observer. Por lo tanto, en la inicialización de las vistas tendremos el siguiente fragmento de código

```
ClockView>>initialize: aModel
    self model: aModel.
    self model addDependent: self.
    ...
```

De esta forma ambas vistas se harán dependientes de las notificaciones hechas por el modelo, sin la necesidad de que el modelo deba conocerlas de forma explícita. Cada vez que la instancia `Clock` realice un cambio hará `#changed:` notificando a todos sus dependientes del cambio ocurrido

```
Clock>>tick
    self time addSeconds:1.
    self changed: #time
```

Este `#changed:` terminará desencadenando un `#update:` en donde se actualizarán las vistas

```
ClockView>>update: aSymbol
    aSymbol = #time
        ifTrue:[self redisplay]
```

Cada subclase definirá el mensaje `#redisplay` de diferente forma, ya que muestran la misma información de maneras distintas. También podría suceder

que alguna subclase no utilice el mismo protocolo para actualizar sus datos y en dicho caso reimplementará el mensaje `#update:`. Luego, una vez que se cierren las vistas, se removerán las dependencias entre la vista y el modelo ya que no son más necesarias

```
ClockView>>close
    ...
    self model removeDependent: self.
    ...
```

Como se vio a lo largo de esta Sección, el patrón Observer es un patrón muy interesante a la hora de desarrollar aplicaciones ya que fomenta la modularidad del diseño, haciéndolo más escalable y sencillo. También se describió la implementación de este patrón en Visualworks, la cual es una implementación simple pero eficaz para la mayoría de las situaciones. Pero existen otros casos por los cuales esta implementación, al ser tan simple, complica el desarrollo. A continuación se describirá una implementación que mejora a la anterior y fue diseñada siguiendo un enfoque orientado a objetos.

B.3. Announcements Framework

La implementación inicial del patrón Observer realizada en Visualworks es simple y fácil de usar, aunque dicho mecanismo del patrón Observer es limitado en cuanto a su implementación. Las notificaciones que son enviadas están representadas mediante símbolos (el símbolo `#value` representa el evento de cambio de valor), pero siendo Smalltalk un lenguaje orientado a objetos puro, sería deseable que un evento sea representado mediante un objeto, el cual pueda tener un estado más complejo del evento ocurrido y la posibilidad de tener comportamiento específico. Por este motivo, un nuevo framework fue creado.

Este framework tiene la siguiente estructura:

Announcement. Esta clase representa a un evento notificado por algún `Announcer` y reemplaza a los símbolos que se usaban en la implementación anterior. No posee ningún protocolo en particular.

Announcer. Es la clase abstracta que representa al Sujeto del patrón Observer. Como se ha explicado en este patrón debe guardar la dependencia de los objetos interesados y definir un protocolo para que sea utilizado por sus subclases.

Observador. El observador es cualquier objeto interesado en recibir una o más notificaciones de algún `Announcer`. Al no necesitar ningún protocolo en particular, puede ser cualquier clase del sistema.

Dentro del protocolo existen varios tipos de mensajes. Si un objecto se encuentra interesado por algún aspecto, debe indicar su interés utilizado algunos de los mensajes provistos, como son el `#when: anAnnouncementOrSymbol send: aSelectorSymbol to: anObject`, `#when: announcementClassOrSet do: aBlock for: anObject` o `#when: anAnnouncementOrSymbol do: aBlock`. Luego, en algún momento el `Announcer` hará alguna modificación y entonces notificará a todos sus dependientes utilizando el mensaje `announce: anAnnouncementClass`. Una vez que el objeto dependiente ya no este interesado puede deshacer este vínculo utilizando el protocolo provisto por `Announcer #unubscribe: aSubscriber from: anAnnouncementClass`.

Un evento que se quiera notificar en este framework, como un cambio en un atributo o un click del mouse, es definido como subclase de la clase abstracta `Announcement`. La subclase puede tener variables de instancia para guardar información adicional, como la hora, o las coordenadas del mouse, o el antiguo valor del parámetro que ha cambiado. Para notificar un evento, el `Announcer` crea y configura una instancia del `Announcement` apropiado y luego todos los observadores reciben una notificación con la instancia del `Announcement`, y al ser este un objeto, cada observador puede consultar su estado, enviarle mensajes y demás.

Retomando el ejemplo del reloj comentado en la sección anterior, consideremos realizarla utilizando `Announcements`. En el momento que se crean las vistas se deberá notificar al modelo que la vista esta interesado en algún cambio:

Si un objecto se encuentra interesado por algún aspecto, debe indicar su interés utilizado el siguiente mensaje

```
ClockView > > initialize: aModel
    self model: aModel .
    self model
            when: TimeChanged
            send: #redisplay
            to: self
```

A partir de ese momento, cada vez que el modelo notifica que ocurrió un TimeChanged en esa instancia de Clock, las vistas recibirán el mensaje `#redisplay:` con la instancia de TimeChanged como argumento. En algún momento la instancia de Clock hará hará alguna modificación y entonces notificará a todos sus dependientes

```
Clock>>tick
    self time addSeconds:1.
    self announce: (TimeChanged new)
```

Con ese único mensaje todos las vistas que eran dependientes son notificadas, sin la necesidad de saber quienes eran ni cuantas eran. Luego, cuando las vistas

ya no estén interesadas pueden deshacer esta dependencia utilizando el protocolo provisto por `Announcer`

```
ClockView >>close
    ...
    self model unubscribe: self from: TimeChanged
    ...
```

A lo largo de esta sección se comentó detalladamente el framework Announcements, mostrando las mejoras en comparación con la implementación original. Para el desarrollo de Meteoroid hemos decidido utilizar este framework, ya que lo consideramos mejor que el original dado que con Announcements cambia el modelo conceptual de los eventos, pasando de un simple símbolo sin variables de instancia ni métodos, a un objeto robusto y con estado. De todas formas, Meteoroid provee ambos mecanismos para poder vincular un modelo con sus vistas.

B.4. ValueModel + Widgets

A la hora de diseñar un conjunto de widgets para crear aplicaciones de escritorio es muy importante tener en cuenta que debe ser simple, pero a la vez completo, reutilizable y escalable.

Un widget es un elemento gráfico que muestra información relevante al usuario, como puede ser un text-area, un input, un list, etc., cuya característica principal es proveer una forma de manipular la información visualizada. En Visualworks, los widgets están altamente acoplados con objetos auxiliares llamados `ValueModel` (VM), para manejar los datos que presenta. En lugar de mantener los datos directamente, un widget delegada el manejo de los datos en su VM. Por lo tanto, cuando un widget recibe datos del usuario, los almacena en su VM y cuando necesita de los datos para actualizar su vista, consulta a su VM. Esto hace que los objetos widgets solo se centren en como mostrar la información y no tanto de donde obtener dicha información de manera actualizada.

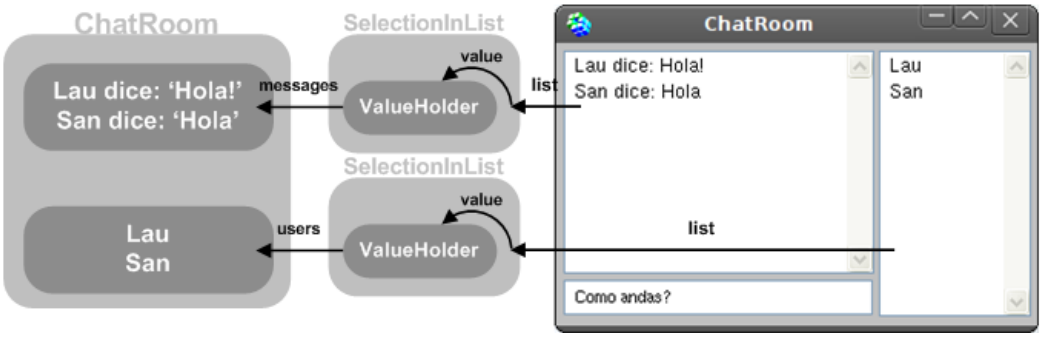

Figura B.4: Asociación entre Widgets y ValueModels

Los VM fueron propuestos por VisualWorks para proporcionar un conjunto uniforme de mensajes para acceder a los datos que se muestran, lo que permite que todos los widgets puedan almacenar y mostrar su información de una forma estándar y simple. El protocolo genérico consta de una variable de instancia `model` el cual esta mirando a cierto aspecto del modelo y de dos mensajes, `#value`, que permite tomar la información asociada al VM, y `#value: anObject`, que cambia el valor del VM y notifica a sus dependientes mediante un `#changed`. Todos los widgets son dependientes de su VM, y por lo tanto esperará que su VM le comunique cuando ha ocurrido algún cambio, para que estos luego soliciten al VM los nuevos datos y poder mostrarlos en su vista correctamente. Por este motivo los widgets pueden ser programados suponiendo que su contenido siempre va a entender el protocolo relacionado a `#value` independientemente del modelo real que tiene los datos.

Existen varias clases relacionadas con este framework, entre las más interesantes y utilizadas se encuentran las siguientes:

ValueHolder. Es la subclase de `ValueModel` más sencilla y más utilizada. Guarda al objeto que representa en una variable de instancia, manejándolo con el protocolo descrito en ValueModel. Por ejemplo, si uno hiciera

```
| valueHolder |
valueHolder := ValueHolder with: 3.
...
```

luego podría acceder a dicho valor haciendo

```
valueHolder value
```

y modificarlo haciendo

```
valueHolder value: 8
```

AspectAdaptor. A diferencia de un ValueHolder, el AspectAdaptor no contiene en su variable de instancia al objeto al cual enmascara, sino que posee dos variables. En la primera guarda un objeto y en la segunda el aspecto de ese objeto que le interesa. Por lo tanto, tiene una indireccion más a la hora de acceder a la información, pero para el exterior seguirá cumpliendo con el protocolo de ValueModel. Además un AspectAdaptor se hace dependiente del aspecto que le interesa del modelo, y notificará con `#value` a todos sus dependientes cuando el modelo notifique un cambio de su aspecto.

Esto es interesante cuando se tiene un modelo de dominio que posee múltiples aspectos, y el valor de cada uno de estos se almacena en una variable de

instancia. El programador puede utilizar mensajes propios para almacenar y recuperar los valores.

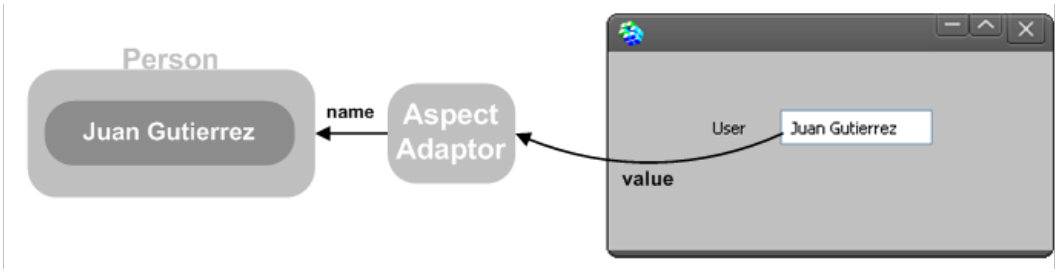

Figura B.5: Funcionamiento de un AspectAdaptor

Supongamos una instancia de Person, que contiene la información normal de cualquier objeto que representa una persona. Uno podría crear un AspectAdaptor de la siguiente manera

```
| aspectAdaptor |
aspectAdaptor := (AspectAdaptor
                    subject: self model)
                        forAspect: #name.
...
```

provocando que esta instancia de AspectAdaptor este interesada en el aspecto `#name`. Luego podría acceder a dicho valor haciendo

```
aspectAdaptor value
```

y el AspectAdaptor se encargará automáticamente de recuperar el nombre del modelo. Algo similar ocurrirá con el mensaje `value:` y con las notificaciones de ese aspecto, ya que cada vez que en el modelo se llame a

```
Person>>name: aString
    self name: aString.
    self changed: #name.
```

el AspectAdaptor, que es dependiente de esa instancia de Person, notificará a todos sus dependientes con el símbolo `#value`.

SelectionInList. Es utilizado para enmascarar una lista y su selección actual. Posee dos aspectos interesantes, `list` y `selectionIndex`, y notificará a sus dependintes cuando cualquiera de sus dos aspectos cambie. Este objeto es una composicion de dos ValueModels simples (`listHolder` y

`selectionIndexHolder`), uno por cada aspecto. Cabe destacar que posee un protocolo genérico aunque este sea distinto a los otros ValueModels, en vez de usar la familia de protocolo `value`, utiliza `list` para modificar y notificar cambios de la lista, y `selectionIndex` para modificar y notificar cambios en la selección de la lista.

Por ejemplo, podemos tener un SelectionInList enmascarando a una lista de personas de la siguiente manera

```
selectionInList := (SelectionInList
                        with: self model
                            peopleList)
                              selectionIndex: 1.
```

y de esta forma se podrá mantener la selección y la lista en sí, utilizando un protocolo simple y genérico.

Como se vio a lo largo de esta Sección, al proveer una forma genérica, simple y muy eficaz de acceder a cualquier aspecto del modelo de dominio, un desarrollador de la aplicación puede crear interfaces gráficas sin preocuparse en exceso del modelo, solo debe asociar los aspectos del modelo con los ValueModels y estos ValueModels con los widgets y todas las aplicaciones se actualizarán automáticamente.

Bibliografía

[1] Dave Crane y Phil McCarthy. Comet and reverse ajax. 2008.

[2] Stéphane Ducasse, Adrian Lienhard y Lukas Renggli. Seaside - A Multiple Control Flow Web Application Framework. 2004.

[3] Hypertext transfer protocol. HTTP/1.1 http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.4.

[4] James Duncan Davidson, Danny Coward. Java servlet specification (specification) version: 2.2 final release. sun microsystems. pages 43–46, 1999.

[5] Página oficial de Cincom VisualWorks. http://www.cincomsmalltalk.com

[6] Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php

[7] Netscape. Client-Side JavaScript Reference. http://docs.sun.com/source/816-6408-10/contents.htm

[8] Glenn E. Krasner y Stephen T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system.

[9] Lista de elementos posibles de HTML. http://www.w3.org/TR/html4/index/elements.html

[10] Origen del término Comet. http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser

[11] Document object model (dom). http://www.w3.org/DOM

[12] Jonathan Snook, Aaron Gustafson, Stuart Langridge y Dan Webb. Accelerated dom scripting with ajax, apis, and libraries. 2007.

[13] James O. Coplien y Douglas C. Schmidt. Pattern languages of program design. páginas 467-494.

[14] DARPA (Defense Advanced Research Project Agency) - 50 Years of Bridging the Gap. 2008.

[15] Bolt Baranek and Newman Inc. A History of the ARPANET: The First Decade 1981.

[16] W. Richard Stevens. TCP/IP Illustrated, Vol. 1: The Protocols.

[17] Headings. http://www.w3.org/TR/html401/struct/global.html#h-7.5.5

[18] CSS versión 1. http://www.w3.org/TR/CSS1

[19] CSS versión 2. http://www.w3.org/TR/CSS2

[20] Página oficial de ECMA. http://www.ecma-international.org

[21] World Wide Web Consortium. http://www.w3.org/Consortium

[22] Thomas Connolly. Database Systems (2ª edición). Página 383.

[23] Hubert Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Transactions on Communications, vol. 28, no. 4, páginas 425 - 432.

[24] Página oficial de Request for Comments. http://www.ietf.org/rfc.html

[25] Defición de HTTP 0.9. http://www.w3.org/Protocols/HTTP/AsImplemented.html

[26] RFC 2045 - Part One:Format of Internet Message Bodies. http://www.ietf.org/rfc/rfc2045.txt

[27] RFC 2046 - Part Two:Media Types. http://www.ietf.org/rfc/rfc2046.txt

[28] RFC 1945 - HTTP 1.0. www.ietf.org/rfc/rfc1945.txt

[29] Netscape. An exploration of dynamic documents. 1995. Link Original (muerto): http://www.netscape.com/assist/net_sites/pushpull.html Chached reference: http://web.archive.org/web/*/www.netscape.com/assist/net_sites/pushpull.html

[30] RFC 2731 - Tag META. www.ietf.org/rfc/rfc2731.txt

[31] David Flanagan. Javascript:the definitive guide. 2006.

[32] Objeto XMLHTTPRequest. http://www.w3.org/TR/XMLHttpRequest

[33] RFC 147 - Socket. http://www.ietf.org/rfc/rfc147.txt

[34] David Makofske, Michael J. Donahoo y Michael J. Donahoo TCP/IP Sockets in C#: Practical Guide for Programmers. 2004.

[35] Rajkumar Buyya. High Performance Cluster Computing: Architectures and Systems, Vol. 1. 1999.

[36] Tony Bourke. Server Load Balancing. 2001.

[37] Engin Bozdag, Ali Mesbah y Arie van Deursen. A comparison of push and pull techniques for ajax. 2007.

[38] Explicación e historia de Plugin. http://en.wikipedia.org/wiki/Plug-in_(computing)

[39] Página oficial de Flash. http://www.adobe.com/es/products/flashplayer

[40] Página oficial de Silverlight. http://silverlight.net

[41] Página oficial de OpenLaszlo. http://www.openlaszlo.org

[42] W3C Compliant. http://www.w3.org/standards/about.html

[43] Michael Mahemoff. Ajax design pattern. página 113, 2006.

[44] Ka-Ping Yee. Chat using dynamic animated images. http://zesty.ca/chat

[45] Steve D. Pate. UNIX Filesystems: Evolution, Design, and Implementation. página 19. 2003.

[46] Tag Iframe. http://www.w3.org/TR/REC-html40/present/frames.html#edef-IFRAME

[47] Shawn M. Lauriat. Advanced ajax architecture and best practices. página 17.

[48] Server-sent events. http://www.w3.org/TR/eventsource

[49] Objeto Activexobject. http://msdn.microsoft.com/enus/library/7sw4ddf8(VS.85).aspx

[50] Don Box. Essential COM. 1998.

[51] Extensible Markup Language. http://www.w3.org/TR/2008/REC-xml-20081126

[52] Elliotte Rusty Harold y W. Scott Means. XML in a Nutshell, Third Edition. 2004.

[53] RFC 2979 - Firewall. http://www.ietf.org/rfc/rfc2979.txt

[54] Ari Luotonen. Web Proxy Servers (Web Infrastructure Series).

[55] Problema en IE con readyState. http://msdn.microsoft.com/en-us/library/ms534361(v=VS.85).aspx

[56] Página oficial APE. http://www.ape-project.org

[57] Moreno Muffatto. Open Source: A Multidisciplinary Approach. 2006.

[58] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui y Anne-Marie Kermarrec. The many faces of publish/subscribe. 2003.

[59] RFC 4627 - JSON. http://www.ietf.org/rfc/rfc4627.txt

[60] JavaServer Faces. http://java.sun.com/javaee/javaserverfaces

[61] Página oficial Lightstreamer. http://www.lightstreamer.com

[62] Página oficial Meteor. http://meteorserver.org

[63] Página oficial Orbited. http://orbited.org

[64] Allen B. Downey. Python for Software Design: How to Think Like a Computer Scientist.

[65] Página oficial de RabbitMQ. http://www.rabbitmq.com

[66] Página oficial de ActiveMQ. http://activemq.apache.org

[67] Página oficial de Pushlets. http://www.pushlets.com

[68] Budi Kurniawan. Java for the Web with Servlets, JSP, and EJB.

[69] Just van den Broecke. Pushlets - Whitepaper

[70] Stéphane Ducasse, Adrian Lienhard y Lukas Renggli. Seaside - a multiple control flow web application framework.

[71] Página oficial de Swazoo. http://www.swazoo.org

[72] Robert Martin, Dirk Riehle y Frank Buschmann. Pattern Languages of Program Design 3. 1998.

[73] Especificación de Javascript en HTML 4.0. http://www.w3.org/TR/REC-html40/interact/scripts.html

[74] Especificación de Useragent. http://www.w3.org/TR/WAIUSERAGENT

[75] Erich Gamma, Richard Helm, Ralph Johnson y John M. Vlissides. Design patterns: Elements of reusable object-oriented software.

[76] Richard Jones y Rafael D Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. 1996.

[77] Cincom Smalltalk. Application developer's guide.

[78] Jonathan Chaffer y Karl Swedberg. Learning jQuery: Better Interaction Design and Web Development with Simple JavaScript Techniques.

[79] Dave Crane, Bear Bibeault, Tom Locke y Thomas Fuchs. Prototype and scriptaculous in action.

[80] Página oficial dhtmlxGrid. http://dhtmlx.com/docs/products/dhtmlxGrid

[81] Página oficial de Google Wave. http://wave.google.com

[82] Página oficial de Google Docs. http://docs.google.com

[83] Página oficial de Swoopo. http://www.swoopo.com

[84] Página oficial de StuffBuff. http://www.stuffbuff.com

[85] Página oficial de DeACentavos. http://www.deacentavos.com

[86] John Murray. Inside Microsoft Windows CE. 1998.

[87] Frank McPherson. How to Do Everything with Windows Mobile. 2006.

[88] Ed Burnette. Hello, Android: Introducing Google's Mobile Development Platform. 2008.

[89] Ben Morris. The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS. 2007.

[90] Jeff LaMarche, David Mark. Beginning iPhone 3 Development: Exploring the iPhone SDK. 2009.

[91] Andrés Fortier, Cecilia Challiol, Juan Lautaro Fernández, Santiago Robles, Gustavo Rossi y Silvia Gordillo. Exploiting Personal Web Servers for Mobile Context-Aware Applications

[92] Anind Kumar Dey. Providing architectural support for building contextaware applications. PhD thesis, Georgia Institute of Technology. 2000.

[93] Cincom Smalltalk. GUI Developer's Guide.

[94] Página oficial de Mozilla prism. http://labs.mozilla.com/2007/10/prism

[95] Licencia MIT. http://www.opensource.org/licenses/mit-license.php

[96] David Sklar. Learning PHP5. 2004.

[97] William E. Weinman. The CGI Book. 1996.

[98] Página oficial de Squeak. http://www.squeak.org

[99] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou y Marcus Denker. Squeak by Example. 2007.

[100] Página oficial de Dolphin. http://www.object-arts.com

[101] Página oficial de Pharo. http://www.pharo-project.org

[102] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou y Marcus Denker. Pharo by Example.

[103] Leonard Richardson y Sam Ruby. RESTful Web Services. 2007.

[104] Paul Anderson. What is web2.0? ideas, technologies and implications for education. 2007.

[105] Cincom Smalltalk. Source Code Management Guide.

[106] Bryan O'Sullivan, John Goerzen y Don Stewart. Real world Haskell. 2009.